第 1 章 Linux 内核简介

世界各地都有人在钻研 Linux 内核，大多是在写设备驱动程序。尽管每个驱动程序都不一样，而且你还要知道自己设备的特殊性，但是这些设备驱动程序的许多原则和基本技术技巧都是一样的。通过本书，可以学会写自己的设备驱动程序，并且可以钻研内核的相关部分。本书涉及到的是设备无关编程技巧，不会将例子跟特殊设备绑定在一起。
本章没有实际编写代码。但我要介绍一些关于 Linux 内核的背景概念，这样到我们稍后开始介绍实际编程时，就很顺利了。
当你学习编写驱动程序的时候，你也会发现很多关于 Linux 内核的知识，这对理解你机器怎么工作很有帮助，并且还可以知道为什么你的机器没有希望的那么快，或者为什么不按照你象要它做的那样做。我们会逐渐介绍一些新概念，先从简单的驱动程序开始，每介绍一些新概念都会看到相关例子代码，这些代码都不需要特殊硬件。

驱动程序作者的作用

作为一个程序员，你可以选择自己的驱动程序，在编程所需时间和结果的灵活性之间做个可以接受的权衡。尽管说驱动程序的灵活性看起来有那么点怪，我喜欢这个词是因为它强调了设备驱动程序提供的是机制，而不是策略。
机制和策略之间的差别是 Unix 设计背后最好的点子之一。实际编程中遇到的大多数问题都可以被划分成两个部分：“ 需要作什么 ”（机制）和“ 这个程序怎么用 ”（策略）。如果这两个主题是由程序不同部分来承担的，或者是由不同的程序组合一起承担的，那么这个软件包就很容易开发，也很适合特殊需求。
举个例子，Unix 的图形显示管理在 X 服务器和窗口管理器之间划了一道线，X 服务器了解硬件并给用户程序提供唯一的接口，而窗口管理器实现特殊的策略并不需要知道硬件的任何信息。人们可以在不同硬件上使用同样的窗口管理器，并且不同用户在同一台工作站上可以使用不同的设置。另一个例子是 TCP/IP 的网络分层结构：操作系统提供抽象的套接字操作，是设备无关的，不同服务器主管这个服务。另外，ftpd 服务器提供文件传输机制，而用户可以使用任何客户端程序；命令行的客户端和图形化界面的客户端都存在，并且谁都可以为传输文件写一个新的用户界面。
只要涉及到驱动程序，就会运用这样的功能划分。软盘驱动程序是设备无关的——这不仅表现在磁盘是一个连续读写的字节数组上。如何使用设备是应用程序要做的事：tar 要连续地写数据，而 mkfs 则为要安装的设备做准备工作，mcopy 依赖于设备上存在的特殊数据结构。
在写驱动程序时，程序员应该特别留心这样的基本问题：我们要写内核代码访问硬件，但由于不同用户有不同需要，我们不能强迫用户采用什么样的特定策略。设备驱动程序应该仅仅处理硬件，将如何使用硬件的问题留给应用程序。如果在提供获得硬件能力的同时没有增加限制，我们就说驱动程序是灵活的。不过，有时必须要作一些策略决策。
可以从不同侧面来看你的驱动程序：它是位于应用层和实际设备之间的软件。驱动程序的程序员可以选择这个设备应该怎样实现：不同的驱动程序可以提供不同的能力，甚至相同的设备也可以提供不同能力。实际驱动程序设计应该是在众多需求之间的一个平衡。例如，不同程序可以同时使用同一个设备，而驱动程序的开发者可以完全自由地决定如何处理同步机制。你可以实现到设备上的内存映射，而完成独立于硬件的具体能力，或者你可以提供给用

户函数库，帮助应用程序的程序员在可用原语的基础上实现新策略，或者诸如此类的方法。一个很重要需要考虑的问题就是，如何在提供给用户尽可能多的选项，平衡你需要编写所花费的时间，以及为使错误尽可能少而保持代码简单之间的平衡。
如果即为同步又为异步操作设计驱动程序，如果允许同时打开多次，并且如果能够发掘所有硬件功能，而不用增加软件层“ 去简化事情 ”——例如将二进制数据转换成文本或者策略相关的操作——那就很容易编写而且很好维护了。达成“ 策略无关 ”实际上是软件设计的共同目标。
实际上，大多数设备驱动程序是和用户程序一起发布的，这些程序可以帮助完成对目标设备的配置和访问。这些程序可以是从简单的配置程序到完整的图形应用。通常还要提供一个客户端库文件。
本书讨论范围是内核，所以我们不考虑策略问题，也不考虑应用程序或支持库。有时，我们会讨论不同策略，以及如何支持这些策略，但我们不会深入到使用一定策略或设备编程需要的细节问题。不过你应该可以理解，用户程序是一个软件包的内核，就算策略无关的软件包也会和配置文件一起发布，这些文件提供了基本机制上的缺省行为。

划分内核

在 Unix 系统中，若干并发进程会参加不同的任务。每个进程都要求获得系统资源，可以是计算、内存、网络连接或别的资源。内核是一整块可执行代码，用它来负责处理所有这样的请求。尽管在不同的内核任务之间的区别不是总能清楚地标识出来，内核的作用还是可以被划分的。如图 1-1 所示，划分为如下这些部分：

进程管理

内核负责创建和终止进程，并且处理它们和外部世界的联系（输入和输出）。对整个系统功能来讲，不同进程之间的通信（通过信号，管道，进程间通信原语）是基本的，这也是由内核来处理的。另外，调度器，可能是整个操作系统中最关键的例程，是进程管理中的一部分。更广广义的说，内核的进程管理活动实现了在一个 CPU 上多个进程的抽象概念。

内存管理

计算机内存是主要资源，而使用内存的策略是影响整个系统性能的关键。内核为每个进程在有限可利用的资源上建立了虚拟地址空间。内核不同部分通过一组函数与内存管理子系统交互，这些包括从简单的 malloc/free 到更稀奇古怪的功能。
（图 1-1）

文件系统

Unix 系统是建立在文件系统这个概念上的；Unix 里几乎所有东西都可以看作文件。内核在非结构的硬件上建立了结构化的文件系统，这个抽象的文件被系统广泛应用。另外，Linux 支持多文件系统类型，即，物理介质上对数据的不同组织方法。

设备控制

几乎每种系统操作最后都要映射到物理设备上。除了处理器，内存和少数其他实体外，几乎所有设备的控制操作都由设备相关的代码来实现。这些代码就是设备驱动程序。内核必须为

每个外部设备嵌入设备驱动程序，从硬盘驱动器到键盘和磁带。内核的这方面功能就是本书的着眼点。

网络

网络必须由操作系统来管理，由于大多数网络操作不是针对于进程的：接收数据包是异步事件。数据包必须在进程处理它们以前就被收集，确认和分发。系统通过程序和网络接口发送数据包，并且应该可以正确地让程序睡眠，并唤醒等待网络数据的进程。另外，所有路由和地址解析问题是在内核里实现的。
在本书结尾部分的第 16 章“ 内核源码的物理布局 ”里，您可以看到 Linux 内核的路标，但现在这里的话应该足够了。
Linux 的一个很好的特征就是，它可以在运行的时候扩展内核代码，也就是说在系统运行的时候你可以增加系统的功能。
每个可以增加到内核中的代码称为一个模块。Linux 内核支持相当多的模块的类型（或‘ 类 ”），但不仅仅只局限于设备驱动程序。每个模块由目标代码组成（没有连接成完整的可执行文件），通过 insmod 程序它们可以动态连接到运行着的内核中，而通过 rmmod 程序就可以去除这些模块。
在图 1-1 中，你可以标别出处理不同任务的不同模块类别——根据模块提供的功能，每个模块属于一个特定的类。

设备类和模块

在 3 类设备中，Unix 看待设备的方式有所区别，每种方式是为了不同的任务。Linux 可以以模块的形式加载每种设备类型，因此允许用户在最新版本的内核上实验新硬件，跟随内核的开发过程。
一考虑到模块，每个模块通常只实现一个驱动程序，因此是可以分类的。例如，字符设备模块，或块设备模块。将模块分成不同的类型或类并不是固定不变的；程序员可以选择在单独一整块代码中创建一个模块实现不同的驱动程序。不过好的程序员会为他们实现的每一个新功能创建不同模块。
现在回到驱动程序，有如下三种类型：

字符设备

可以象文件一样访问字符设备，字符设备驱动程序负责实现这些行为。这样的驱动程序通常会实现 open，close，read 和 write 系统调用。系统控制台和并口就是字符设备的例子，它们可以很好地用流概念描述。通过文件系统节点可以访问字符设备，例如/dev/tty1 和/dev/lp1。在字符设备和普通文件系统间的唯一区别是：普通文件允许在其上来回读写，而大多数字符设备仅仅是数据通道，只能顺序读写。当然，也存在这样的字符设备，看起来象个数据区，可以来回读取其中的数据。

块设备

块设备是文件系统的宿主，如磁盘。在大多数 Unix 系统中，只能将块设备看作多个块进行访问为，一个块设备通常是 1K 字节数据。Linux 允许你象字符设备那样读取块设备——允许一次传输任意数目的字节。结果是，块设备和字符设备只在内核内部的管理上有所区别，因此也就是在内核/驱动程序间的软件接口上有所区别。就象字符设备一样，每个块设备也

通过文件系统节点来读写数据，它们之间的不同对用户来说是透明的。块设备驱动程序和内核的接口和字符设备驱动程序的接口是一样的，它也通过一个传统的面向块的接口与内核通信，但这个接口对用户来说时不可见的。

网络接口

任何网络事务处理都是通过接口实现的，即，可以和其他宿主交换数据的设备。通常，接口是一个硬件设备，但也可以象 loopback（回路）接口一样是软件工具。网络接口是由内核网络子系统驱动的，它负责发送和接收数据包，而且无需了解每次事务是如何映射到实际被发送的数据包。尽管“ telnet ”和“ ftp ”连接都是面向流的，它们使用同样的设备进行传输；但设备并没有看到任何流，仅看到数据报。

由于不是面向流的设备，所以网络接口不能象/dev/tty1 那样简单地映射到文件系统的节点上。Unix 调用这些接口的方式是给它们分配一个独立的名字（如 eth0）。这样的名字在文件系统中并没有对应项。内核和网络设备驱动程序之间的通信与字符设备驱动程序和块设备驱动程序与内核间的通信是完全不一样的。内核不再调用 read，write，它调用与数据包传送相关的函数。

事实上，Linux中还有一类“ 设备驱动程序模块 ”：SCSI*设备驱动程序。尽管每个连接到SCSI总线上的外设在/dev目录中不是字符设备就是块设备，但软件的内部组织并不完全同。

正如网络接口给网络子系统提供硬件相关的功能一样，SCSI 控制器提供给 SCSI 子系统如何访问实际接口电缆。SCSI 是计算机和外设之间的通信协议，每种 SCSI 设备都响应相同的协议，与计算机插的是哪种控制板没有关系。因此，Linux 内核嵌入一个所谓 SCSI“ 实现 ”（即，文件操作到 SCSI 通信协议的映射）。驱动程序编写人员必须在 SCSI 抽象层和物理电缆之间实现这种映射。这种映射依赖于 SCSI 控制器，却与 SCSI 电缆上连接的设备无关。

除了设备驱动程序，还有一些别的模块化加载到核心中的驱动程序，可以是软件，也可以时硬件。并非实现驱动程序的模块中最重要的一类是文件系统。文件系统类型是与为了表示目录和文件等实体的信息的组织方式相关的。因此这种实体不是“ 设备驱动程序 ”，其中并没有确定某种设备与信息组织的方式有关；由于文件系统将原始数据组织为高层信息，它实际上是软件驱动程序。

如果你考虑到 Unix 系统对底层文件系统的依赖程度，你就可以意识到软件概念对系统操作的重要性。文件系统信息解码的能力位于内核分层的最底层，而且是最重要的；甚至如果给自己的 CD-ROM 写一个新的块设备驱动程序时，要是不能对其上数据上运行 ls 或 cp，那个驱动程序就根本没有什么用处。Linux 支持文件系统模块，它的软件接口声明了可以操作在文件系统节点，目录，文件和超级块上的操作。因此，接口与实际数据传出传入磁盘是完全独立的，这是由块设备驱动程序完成的。对程序员来讲，由于正式内核中已经包含大部分重要文件系统类型的代码，编写一个文件系统模块是很不寻常的要求。

安全问题

最近讨论安全问题很是时髦，而大多数程序员都考虑过系统的安全问题，所以，为防止产生误解，一开始我就要谈论这个问题。

安全性有两方面。一个问题是由于用户对程序的误操作或发掘出错误造成的；另一个问题是由程序员实现的（错误）功能造成的。显然，程序员比普通用户拥有更多的权利。换句话说

* SCSI时Small Computer System Interface（小型计算机系统接口）的缩写；它时工作站市场上形成的标准，也广泛应用在PC领域。

就是，以 root 权限运行从朋友那拿来的程序比给他/她一个 root 外壳要危险得多。尽管访问编译器本质上不是安全性漏洞，但当所编译的代码执行时，还是会出现漏洞；要小心处理模块，因为内核模块可以做任何事。模块比超级用户的外壳的威力还要强大，它的特权是由 CPU 确认的。

所有系统中的安全性检查都是内核代码完成的。如果内核有安全性漏洞，系统就有漏洞。在内核正式发布版本中，只有 root 可以加载模块；系统调用 create_module 检查调用进程的用户 ID。因此在正式内核中，只有超级用户，或是成功地称为 root 的闯入者可以利用特权代码的威力。

幸亏在编写设备驱动程序或别的模块时，很少需要考虑安全性问题，因为访问块设备的进程已经受到更通用的块设备技术的严格控制了。例如，对于块设备来说，安全是由文件系统节点的权限和 mount 命令处理的，因此，在实际的块设备驱动程序中通常没有什么好检查的。尽管如此，从收到第三方软件开始，尤其是当软件涉及到内核时，要格外小心；由于每个人都可以访问源代码，都可以改写和重新编译这些东西。如果可以信任发布版中已经编译好的内核，就要避免编译来源不可靠的内核——如果你不能以 root 身份运行编译好的二进制代码，那最好不要运行编译好的内核。例如，有敌意改动过的内核可能允许任何人加载模块，因此通过 create_module 就可以打开一个不希望出现的后门。

如果确实在模块相关部分考虑到安全性问题，我敦促你看一看 securelevel 内核变量是怎样使用的。当我写这些的时候，Linux 社团里正在讨论控制 securelevel 变量来防止模块加载和卸载。很有趣，你可以注意到最近的内核支持在内核编译时可以删除对模块的支持，这样就关闭了所有安全性有关的漏洞。

版本编号

做为开始研究编程前的最后一点，我很愿意就 Linux 非同寻常的版本编号这个问题加以说明，同时说明本书使用哪个版本的内核。

首先要注意，Linux 系统使用的每个软件包都有它自己的发行号，它们之间有一些内部的依赖关系：你需要在特定版本的软件包上使用特定版本的另一个软件包。Linux 发布版的开发者通常要处理大量软件包的匹配问题，而使用者从预先打包的发布版本上安装软件，无需考虑版本问题。而另一方面，那些更换和升级系统软件的人都要自己处理版本问题。所幸的是，一些最新发布的版本允许单个软件包的升级，它是通过检查软件包间的依赖性实现的，这可大大简化了使用者为维护系统软件一致所要做的事情。

在本书中，我假定你有 2.6.3 版或更新的 gcc 编译器，1.3.57 版或更新版本的模块工具，以及最新的程序开发用 GNU 工具（最重要的是 gmake）。这些要求不是那么严格，由于几乎所有的 Linux 安装版上都配有 GNU 工具，这些版本都相对较旧（此外，内核 2.0 版及其后继版本不能用比版本 2.6 还旧的 gcc 编译）。注意，最近的内核包含一个称为 Documentation/Changes 的文件，它罗列了所有编译和运行这个内核版本所需要的软件。1.2 的源码中没有这个文件。

只要涉及到内核部分，我将集中介绍 2.0.X 和 1.2.13 版本，编写适用于这两个版本的代码。偶数内核版本（如 1.2.x 和 2.0.x）是稳定版本，专门用于发布的版本。而相反，奇数版本是开发中的一个快照，相当短暂；最新的版本代表最新的发展状况，但可能过几天就过时了。这里没有别的通用原因来解释为什么要运行 1.3 和 2.1 核心，除非他们是最新的版本。不过有时你会选择运行一个开发用内核，或时由于它有一些在稳定版本中没有的特性，而你正好需要这些特性，或者很简单，你就是自己改动了这些版本的一些特性，并且你没有时间更新你的补丁。不过还是要注意，实验用内核没有什么保证，如果由于在非当前奇数版本的内核

中的臭虫导致你丢失数据，没人帮得了你。不过，本书支持直到 2.1.43 版的开发用内核，最后一章介绍如何编写可以区分 2.0 和 2.1.x 之间不同之处的驱动程序。
至于 1.2.13，尽管相当旧，我仍觉得这是一个很重要的版本。尽管在某些平台上 2.0.x 比 1.2.13 要快得多，但 1.2.13 非常小，对想使用旧硬件的人来说是个很好的选择。基于 386 处理器，带小 RAM 的低价系统非常适合于做嵌入式系统或自动化控制器，用 1.2.13 比 2.0.x 也许会更快，由于 1.2.13 是以前 1.2.x 版本的错误修订版，我不想考虑更早的 1.2 内核。
无论什么时候，只要 1.2.13 和 2.0 或最新的 2.1 版不兼容，我都会告诉你。
不管怎样，我的主要目标版本是 Linux 2.0，本书介绍的一些特征在旧版本中是没有的。大多数样例模块在很多内核版本中都是可以编译和运行的；特别是它们都已经在 2.0.30 版本上测试过了，而且大多数也都经过了 1.2.13 版本的测试。有时，我的例子不支持 1.2，但只在本书的第二部分中出现这样的情况，这部分对设计来讲更深入，并且可以不用考虑旧版本。
由于 Linux 已不再只是“ PC 兼容机的 Unix 变体 ”了，它的另一个特点是，它是平台独立的操作系统。事实上，除了 x86 以外，它也成功地用于 Axp-Alpha，Sparc 处理器，Mips Rx000 和其他一些平台。本书也尽力达到平台无关性，而且所有的例子代码都在 PC，Alpha 平台和 Sparc 机器上测试过。由于代码在 32 位和 64 位处理机（Alpha）上测试过，它们应该可以在其他平台上编译和运行。正如你所预料的，依赖特殊硬件的编码不能适用于所有的平台，但在源代码中都有所说明。

许可证术语

Linux 是按 GNU“ General Public License ”(GPL)分布许可证的。这是由自由软件基金会为 GNU 计划设计的文档。GPL 允许任何人重新发布，出售 GPL 的产品，只要允许接收者从源码重新建立二进制文件的精确副本。另外，任何从 GPL 产生的软件产品也必须按 GPL 发布。这种许可的主要目的是通过允许每个人随意修改程序来推广知识；同时，向公众出售软件的人仍可以做他们的工作。尽管这是个很简单的目标，还是有一些正在进行的有关 GPL 及其使用的讨论。如果你想读到这些许可证，你可以在你系统的若干个地方找到它们，包括目录 /usr/src/linux，有一个称为 COPING 文件。
当涉及到第三方和定制的模块时，它们不属于内核，因此你无需限制它们使用 GPL 许可证。模块通过明确的接口使用内核，它不是内核的一部分，这种关系和用户程序通过系统调用使用内核很相似。
简而言之，如果你的代码深入内核内部而你又想发布代码，你必须使用 GPL。尽管自己修改自己使用不是非要使用 GPL，如果你要发布代码，就必须在发布中包含源代码——人们获得你的软件包，并且可以随意修改它，重建二进制文件。换句话说，如果你写了一个模块，你就可以按照二进制格式发布你的模块。然而，由于模块通常要针对要连接的内核重新编译（在第 2 章“ 编写和运行模块 ”中的“ 版本相关性 ”一节中，第 11 章“ Kerneld 和高级模块化 ”的“ 模块中的版本控制 ”一节中都有所介绍），这也不总是可行的。对发布模块的二进制代码一般障碍是，模块包含了定义或声明在内核头文件中的代码；不过这个障碍并不能成立，因为头文件是内核公共接口的一部分，因此它不受许可证制约。
至于谈到本书，大部分源码都是可以重新发布的，或者以源码形式，或者以二进制代码形式，不论 O’Reilly 还是我都不对任何基于此的工作有任何许可证权。所有程序都可以通过 FTP 从 ftp://ftp.ora.com/pub/examples/linux/drivers/下载，而且同一个目录下的 LICENCE 文件有具体的许可证说明。
当例子中包含了部分内核代码时，GPL 就适用了：与源码一同发行的文本很清楚地说明了这一点。这仅对某些源文件是有效的，对本书主题而言，这是次要的。

全书概貌

从此开始，我们进入内核编程的世界。第 2 章介绍模块化，解释了这门技艺的秘密，并给出了运行模块的代码。第 3 章，字符设备驱动程序，讨论字符设备驱动程序并且给出了基于内存的设备驱动程序的完整代码，可以按你的喜好进行读写。使用内存做为设备的硬件基础，可以使任何人运行例子代码，而无需增加特殊硬件。
调式技术对程序员来讲是至关重要的，这些内容在第 4 章“ 调试技术 ”中介绍。这样，运用我们新的调试技巧，我们将面对字符设备驱动程序高级功能，如阻塞型操作，select 的使用以及非常常用的 ioctl 调用；这是都是第 5 章“ 字符设备驱动程序的扩展操作 ”的主题。
在涉及硬件管理之前，我们先解剖几个内核软件接口：第 6 章“ 时间流 ”，讲解内核是如何管理时间的，第 7 章“ 获取内存 ”，讲解内存分配。
接下来我们着重于硬件：第 8 章“ 硬件管理 ”，介绍 I/O 端口的管理和设备中的内存缓冲区管理；之后在第 9 章“ 中断处理 ”介绍中断处理。遗憾的是，由于需要某些硬件支持来测试中断的软件接口，不是每个人能运行本章给出的样例代码。我已经尽我全力保持所需的硬件支持减少到最小，但你还得亲自动手用烙铁做你的硬件“ 设备 ”。这个设备仅仅是一个加到并口上的跳线，所以我希望这不是问题。
第 10 章“ 合理使用数据类型 ”又提供一些有关编写内核软件和一致性问题的建议。
在本书的第二部分，我们更加雄心勃勃；因此从第 11 章开始，我们重新讨论模块化，更加深入讨论这个问题。
第 12 章“ 加载块设备驱动程序 ”介绍了如何实现块设备驱动程序，强调和字符设备驱动程序的区别。接下来，第 13 章“ Mmap 和 DMA ”讲解了我们原先在内存管理中留下来的问题：mmap 和 DMA。到此为止，关于字符设备和块设备驱动程序的所有问题我们都介绍过了。
接下来介绍第三类设备驱动程序：第 14 章“ 网络设备驱动程序 ”讨论一些关于网络接口的细节，剖析了样例网络设备驱动程序的代码。
有些设备驱动程序的功能直接依赖于外设所在的接口总线，所以第 15 章“ 外设总线概貌 ”介绍了现在经常用到的总线实现的主要功能，着重介绍内核支持的 PCI 总线。
最后，第 16 章是内核源代码的一次检阅：对那些想理解全部设计的人来讲，这是一个起点，但他们可能会被 Linux 浩如烟海的代码吓倒。
在 Linux2.0 版发布后不久，2.1 开发树开始引入不兼容性；这是在第一个月中引入的最重要的内容。第 17 章“ 近期发展 ”，它几乎可以看作是附录，它收集所有在 2.1.43 版本发布之前不兼容的东西，并且提供了解决这些兼容性问题的方法。在这章的最后，你可以编写出一个设备驱动程序，它能够在 1.2.13 版本上编译，运行，也可以在所有 2.0 和 2.1.43 版本之间的内核上编译，运行。2.2 很有希望会和 2.1.43 非常相似，你的软件需要为此做好准备。

第 2 章 编写和运行模块

非常高兴现在终于可以开始编程了。本章将介绍模块编程和内核编程所需的所有必要的概念。我们将要不多的篇幅来编写和运行一个完整的模块。这种专业技术（expertise）是编写如何模块化设备驱动程序的基础。为了避免一下子给你很多概念，本章仅介绍模块，不介绍任何类别的设备。

这里介绍的所有内核内容（函数，变量，头文件和宏）也将在本章最后的参考部分再次介绍。

如果你已经座不住了，下面的代码是一个完整的“ Hello, World ”模块（这个模块事实上并没什么功能）。它可以在Linux 2.0 或以上版本上编译通过，但不能低于或等于 1.2，关于这一点本章将在稍后的部分解释*。

（代码）

函数 *printk* 是由 Linux 内核定义的，功能与 *printf* 相似；模块可以调用 printk，这是因为在 *insmod* 加载了模块后，模块就被连编到内核中了，也就可以调用内核的符号了。字符串<1>是消息的优先级。我之所以在模块中使用了高优先级是因为，如果你使用的是内核 2.0.x 和旧的 *klogd* 守护进程，默认优先级的消息可能不能显示在控制台上（关于这个问题，你可以暂且忽略，我们将在第 4 章，“ 调试技术 ”，的“ Printk ”小节中详细解释）。

通过执行 *insmod* 和 *rmmod* 命令，你可以试试这个模块，其过程如下面的屏幕输出所示。注意，只有超级用户才能加载和卸载模块。

（代码）

正如你所见，编写一个模块很容易。通过本章我们将深入探讨这个内容。

模块与应用程序

在深入探讨模块之前，很有必要先看一看内核模块与应用程序之间的区别。

一个应用从头到尾完成一个任务，而模块则是为以后处理某些请求而注册自己，完成这个任务后它的“ 主 ”函数就立即中止了。换句话说就是，*init_module()*（模块的入口点）的任务就是为以后调用模块的函数做准备；这就好比模块在说，“ 我在这，这是我能做的。”模块的第二个入口点，*cleanup_module*，仅当模块被下载前才被调用。它应该跟内核说，“ 我不在这了，别再让我做任何事了。”能够卸载也许是你最喜爱的模块化的特性之一，它可以让你减少开发时间；你无需每次都花很长的时间开机关机就可以测试你的设备驱动程序。

作为一个程序员，你一定知道一个应用程序可以调用应用程序本身没有定义的函数：前后的连编过程可以用相应的函数库解析那些外部引用。*printf* 就是这样一个函数，它定义在 libc 中。然而，内核要仅能连编到内核中，它能调用的仅是由内核开放出来的那些函数。例如，上面的 *helllo.c* 中的 *printk* 函数就是内核版的 *printf*，并由内核开放给模块给使用；除了没有浮点支持外，它和原函数几乎一模一样。

如图 2-1 所示，它勾画了为了在运行的内核中加入新函数，是如何调用函数以及如何使用函数指针的。

由于没有库连接到模块中，源码文件不应该模块任何常规头文件。与内核有关的所有内容都定义在目录*/usr/include/linux* 和*/usr/include/asm* 下的头文件中。在编译应用程序也会间接使

* 正如第 1 章，“ Linux内核简介 ”，中所述，这个例子和本书中的所有其他例子都可以从O’Reilly的FTP站点上下载。

用这些头文件；其中的内核代码通过#ifdef __KERNEL__保护起来。这两个内核头文件目录通常都是到内核源码所在位置的符号连接。如果你根本就想要整个内核源码，你至少还要这两个目录的头文件。在比较新的内核中，你还可以在内核源码中发现 net 和 scsi 头文件目录，但很少有模块会需要这两个目录。
内核头文件的作用将稍后需要它们的地方再做介绍，
内核模块与应用程序的另一个区别是，你得小心“ 名字空间污染 ”问题。程序员在写小程序时，往往不注意程序的名字空间，但当这些小程序成为大程序的一部分时就会造成许多问题了。名字空间污染是指当存在很多函数和全局变量时，它们的名字已不再富有足够的意义来很容易的区分彼此的问题。不得不处理这种应用程序的程序员必须花很大的精力来单单记住这些“ 保留 ”名，并为新符号寻找新的唯一的名字。如果在写内核代码时出现这样的错误，这对我们来说是无法忍受的，因为即便最小的模块也要连编到整个内核中。防止名字空间污染的最佳方法是把所有你自己的符号都声明为 static 的，而且给所有的全局量加一个 well-defined 前缀。此外，你还可以通过声明一个符号表来避免使用 static 声明，这些内容将在本章的“ 注册符号表 ”小节中介绍。即便是模块内的私有符号也最好使用选定的前缀，这样有时会减轻调试的工作。通常，内核中使用的前缀都是小写的，今后我们将贯彻这一约定。
内核编程和应用程序编程的最后一个区别是如何处理失效：在应用程序开发期间，段违例是无害的，利用调试器可以轻松地跟踪到引起问题的错误之处，然而内核失效却是致命的，如果不是整个系统，至少对于当前进程是这样的。我们将在第 4 章“ 调试系统失效 ”小节中介绍如何跟踪内核错误。

用户空间和内核空间

本节的讨论概而言之就是，模块是在所谓的“ 内核空间 ”中运行的，而应用程序则是在“ 用户空间 ”中运行的。这些都是操作系统理论的最基本概念。
事实上，操作系统的作用就是给程序提供一个计算机硬件的一致的视图。此外，操作系统处理程序的独立操作，并防止对资源的未经授权的访问。当且仅当 CPU 可以实现防止系统软件免受应用软件干扰的保护机制，这些不同寻常的工作才有可能实现。
每种现代处理器都能实现这种功能。人们选择的方案是在 CPU 内部实现不同的操作模式（或级）。不同的级有不同的作用，而且某些操作不允许在最低级使用；程序代码仅能通过有限数目的“ 门 ”从一个级切换到另一个级。Unix 系统就是充分利用这一硬件特性设计而成的，但它只使用了两级（与此不同，例如，Intel 处理器就有四级）。在 Unix 系统中，内核在最高级执行（也称为“ 管理员态 ”），在这一级任何操作就可以，而应用程序则执行在最低级（所谓的“ 用户态 ”），在这一级处理器禁止对硬件的直接访问和对内存的未授权访问。
正如前面所述，在谈到软件时，我们通常称执行态为“ 内核空间 ”和“ 用户空间 ”，它们分别引用不同的内存映射，也就是程序代码使用不同的“ 地址空间 ”。
Unix 通过系统调用和硬件中断完成从用户空间到内核空间的控制转移。执行系统调用的内核代码在进程的上下文上执行——它代表调用进程操作而且可以访问进程地址空间的数据。但与此不同，处理中断的代码相对进程而言是异步的，而且与任何一个进程都无关。
模块的功能就是扩展内核的功能；运行在内核中的模块化的代码。通常，一个设备驱动程序完成上面概括的两个任务 :模块的某些函数做为系统调用执行，而某些函数则负责处理中断。

内核中的并发

内核编程新手首先要问的问题之一就是多任务是如何管理的。事实上，除了调度器之外，关于多任务并没有什么可以多说的，而且调度器也超出了程序员的一般活动范围。你可能会遇到这些任务，除了掌握如下这些原则外，模块编写者无需了解多任务。
与串行的应用程序不同，内核是异步工作的，代表进程执行系统调用。内核负责输入/输出以及系统内对每一个进程的资源管理。
内核（和模块）函数完全在一个线程中执行，除非它们要“睡眠”，否则通常都是在单个进程的上下文中执行——设备驱动程序应该能够通过交织不同任务的执行来支持并发。例如，设备可能由两个不同的进程同时读取。设备驱动程序串行地响应若干 read 调用，每一个都属于不同的进程。由于代码需要区别不同的数据流，内核（以及设备驱动程序）必须维护内部数据结构以区分不同的操作。这与一个学生学习交织在一起的若干门课程并非不无相似之处：每门课都有一个不同的笔记本。解决多个访问问题的另一个方法就是避免它，禁止对设备的并发访问，但这种怠惰的技术根本不值的讨论。
当内核代码运行时，上下文切换不可能无意间发生，所以设备驱动程序无需是可重入的，除非它自己会调用 *schedule*。必须等待数据的函数可以调用 *sleep_on*，这个函数接着又调用 *schedule*。不过你必须要小心，存在某些函数会无意导致睡眠，特别是任何对用户空间的访问。利用“天然非抢占”特性不是什么好的方法。我将在第 5 章，“字符设备驱动程序的扩展操作”的“编写可重入代码”小节中讲解可重入函数。
就对设备驱动程序的多个访问而言，有许多不同的途径来分离这些不同的访问，但都是依赖于任务相关的数据。这种数据可以是全局内核变量或是传给设备驱动程序函数的进程相关参数。最重要的用来跟踪进程的全局变量是`current`：一个指向`struct task_struct`结构的指针，在`<linux/sched.h>`中定义。current指针指向当前正在运行的用户进程。在系统调用执行期间，如*open*或*read*，当前进程就是调用这个调用的进程[*]。如果需要的话，内核代码就可以利用`current`使用进程相关信息。第 5 章“设备文件的访问控制”小节中就有使用这种技术的例子。
编译器就象外部引用 printk 一样处理`current`。模块可以在任何需要的地方引用`current`，*insmod*会在加载时解析出所有对它的引用。例如，如下语句通过访问`struct task_struct`中的某些域打印当前进程的进程 ID 和命令名：
（代码）
存储在`current->comm`中的命令名是当前进程最后执行的可执行文件的基名。

编译和加载

本章的剩下部分将介绍编写虽然是无类别但很完整的模块。就是说，模块不属于任何第 1 章“设备和模块的类别”中罗列的类别中的任何一个。本章中出现的设备驱动程序称为*skull*，是“Simple Kernel Utility for Loading Localities”的缩写。去掉这个模块提供的范例函数，你可以重用这个模块，向内核加载你自己的本地代码。[*]
在我们介绍 *init_module* 和 *cleanup_module* 的作用之前，首先让我们写一个 *Makefile* 来编译内核可以加载的目标代码。

[*] 在版本 2.0 中，为了支持SMP，`current`是一个宏，扩展为`current_set[`*`this_cpu`*`]`。优化了对`current`访问的 2.1.37 则将它的值存放在堆栈中，也就去掉了全局符号。

首先，在包含任何头文件前，我们需要在预处理器中定义符号__KERNEL__。这个符号用于选择使用头文件的哪一部分。由于*libc*包含了这些头文件*，应用程序最终也会包含内核头文件，但应用程序不需要内核原型。于是就用__KERNEL__符号和#ifdef将那些额外的去掉。将内核符号和宏开放给用户空间的程序会造成那个程序的名字空间污染。如果你正在为一台SMP（对称多处理器）机器编译，你还需要在包含内核头文件前定义__SMP__。这一要求似乎有点不那么方便，但一旦开放人员找到达成SMP透明的正确方法，它就会逐渐消失的。另一个很重要的符号就是 MODULE，必须在包含<linux/module.h>前定义这个符号。除非要把设备驱动程序编译到内核映象中去，MODULE 应该总是定义了的。由于本书所涉及的驱动程序都不是直接连编到内核中去的，它们都定义了这个符号。

由于头文件中的函数都是声明为inline的，模块编写者还必须给编译器指定-*O*选项。*gcc*只有打开优化选项后才能扩展内嵌函数，不过它能同时接受-*g*和-*O*选项，这样你就可以调试那些内嵌函数的代码了*。

最后，为了防止发生令人不愉快的错误，我建议你使用-*Wall*（全面报警）编译选项，并且还要修改源码去除所有编译器给出的警告，即便这样做会改变你已有的编程风格，你也要这么做。

所有我目前介绍的定义和选项都在 *make* 使用的 CFLAGS 变量中。

除了一个合适的 CFLAGS 变量外，将要编写的 *Makefile* 还需要一个将不同目标文件连接在一起的规则。这条规则仅当一个模块被分成若干个不同的源文件时才需要，这种并非很不常见。通过命令 ld -r 将模块连接在一起，这条命令虽然调用了连接器，但并没有连编操作。这是因为输出还是一个目标文件，它是输入文件的混合。-r 选项的意思是“ 可重定位 ”； 输出文件是可重定位的，这是因为它尚未嵌入绝对地址。

下面的 Makefile 实现了上述的所有功能，它能建立由两个源文件组成的模块。如果你的模块是由一个源文件组成的，只要跳过包含 ld -r 的那项就可以了。

（代码）

上面文件中那个复杂的 install 规则将模块安装到一个版本相关的目录中，稍后将做解释。Makefile 中的变量 VER 是从<linux/version.h>中截取的版本号。

模块编好了，接下来必须把它加载到内核中。正如我前面所说，insmod 就是完成这个工作的。这个程序有点象 ld，它要将模块中未解析的符号连编到正在运行的内核的符号表中。但与连接器不同，它并不修改磁盘文件，而是修改内存映象。insmod 有很多命令选项（如果想知道细节，可以看 man），可以在模块连编到内核前修改模块中的整数值和字符串值。因此，如果一个模块设计得体，可以在加载时对其进行配置；加载时配置要比编译时配置更灵活，但不幸的是，有时候仍然有人使用后者。加载时配置将在本章的后面“ 自动和手动配置 ”小节中讲解。

感兴趣的读者可能想知道内核是怎样支持 insmod 的：它依赖于 kernel/modulec.c 中定义的几个系统调用。sys_create_module 为装载模块分配内存（这些内存是由 vmalloc 分配的，见第 7 章“ 获取内存 ”中的“ vmalloc 及其同胞 ”一节），为了连编模块，系统调用 get_kernel_syms 返回内核符号表，sys_init_module 将可重定位目标码复制到内核空间并调用模块的初始化函数。

如果你看过了内核源码，你就会发现系统调用的名字都有 sys_前缀。所有系统调用都是这样，

---

* 我这里使用了“ 本地 ”，它是指个人对系统的修改，套用了Unix古老而优秀的传统*/usr/local*。

* 对于版本 5 和以往的版本来说是这样的。对于版本 6（*glibc*）来说可能会发生变化，但在我写这本书时讨论尚未结束。

* 不过你要注意，使用任何超过-O2 的优化选项后，编译器可能会把源码中未声明为inline的函数也按内嵌处理，这样做是非常危险的。对于内核代码来说，某些函数被调用时需要有一个标准的堆栈框架，这种优化就会导致问题出现。

其他函数并没有这个约定；当你在源码中查找系统调用时，知道这一点会对你有所帮助。

版本相关性

要时刻牢记，对于你想连编的每一个不同版本的内核，你的模块都要相应地编译一次。每个模块都定义了一个称为 `kernel_version` 的符号，*insmod* 检查这个符号是否与当前内核版本号匹配。较新的内核已在`<linux/module.h>`中替你定义了这个符号（这也就是为什么 *hello.c* 中没有对它的声明）。这也意味着，如果你的模块是由多个源文件组成的，你只能有一个源文件包含了<linux/module.h>。与此相反，当你在 Linux 1.2 下编译时，必须在你的源码中定义 kernel_version。

如果版本不匹配，而你仍然想在不同版本的内核里加载你的模块，可以在 insmod 命令中指定-f（“强制”）选项完成，但这个操作不安全，可能会失败。而且很难事先说明要发生那种情况。由于符号不匹配，加载就会失败，此时你会得到一个错误信息。内核内部的变化也会造成加载失败。如何这种情况发生了，你可能会在系统运行时得到一个非常严重的错误，很可能造成系统 panic——出于这个缘由，注意版本失配。事实上，通过内核里的“版本机制”更完美地解决版本失配问题（稍后，第 11 章“Kerneld 和高级模块化”的“模块内版本控制”小节将介绍这一更先进的内容）。

如果你需要为某个特定的内核编译模块，你必须在上面的 Makefile 中包含相应内核的头文件（例如，通过声明不同的 INCLUDEPATH）。

为了处理加载时的版本相关性，insmod 安装特定的路径查询：如果不能在当前目录找到模块，就在版本相关的目录中查找，如果还失败就在/lib/modules/misc 中查找。上面那个 Makefile 中的 install 规则就遵循了这一约定。

写一个可以在从 1.2.13 到 2.0.x 的任一版本的内核上编译的内核是件复杂的任务。模块化接口已经做了修改，配置越来越容易。你可以看到上面的那个 hello.c 中，只要你只处理较新的内核就什么都不用声明。与此不同，可移植的接口如下所示：

（代码）

在 2.0 或更新的内核中，module.h 包含了 version.h，而且，如果没有定义__NO_VERSION__，module.h 还定义了 kernel_version。

如果你需要将多个源文件连接在一起组成一个模块，而又有多个文件都需要包含<linux/module.h>——比如你需要 module.h 里声明的宏，就可以使用符号__NO_VERSION__。在包含 module.h 前定义__NO_VERSION__就可以在你不想要自动声明字符串 kernerl_version 的源文件里防止它的发生（ld -r 会对一个符号的多处定义报警）。本书中的模块就使用__NO_VERSION__达成这一目的。

其他基于内核版本的相关性可以通过预处理的条件编译解决——version.h定义了整数宏 LINUX_VERSION_CODE。这个宏展开后是内核版本的二进制表示，一个字节代表版本发行号的一部分。例如，1.3.5 的编码是 66309（即，0x10305）。*利用这个信息，你可以轻松地判断你正处理的是哪个版本的内核。

当你检查某个版本时，使用十进制表示是不方便的。为了在一个源文件里支持多个内核版本，我将用下面的宏通过版本号的 3 个部分构建版本编码：

（代码）

* 这样就可以在稳定版本间存在 256 之多的开发用版本。

内核符号表

我们已经知道 insmod 是如何利用公开内核符号来解析未定义符号的了。这张表包含了实现模块化设备驱动程序所需的全局内核项——函数和变量。可以从文件/proc/ksyms 中以文本的方式读取这个公开符号表
当你的模块被加载时，你声明的任何全局符号都成为内核符号表的一部分，你可以从文件/proc/ksyms 或命令 ksyms 的结果了解这一点。
新模块可以使用你开放出来的符号，而且你在其他模块之上堆叠新模块。在主流的内核源码中也使用了这种模块堆叠的方法：msdos 文件系统依赖于 fat 模块开放出来的符号，而 ppp 驱动程序则堆叠在报头压缩模块上。
在处理复杂对象时，模块堆叠非常有用。如果以设备驱动程序的形式实现一个新的抽象，它可以提供一个设备相关的插接口。比如，帧缓冲视频驱动程序可以将符号开放给下层 VGA 驱动程序使用。每个用户都加载帧缓冲视频驱动程序，然后在根据自己安装的设备加载相应的 VGA 模块。
分层次的模块化简化了每一层的任务，大大缩减了开发时间。这同我们第 1 章中讨论的机制与策略分离很相似。

注册符号表

另一种开放你的模块中的全局符号的方法是使用函数 register_symtab，这个函数是符号表管理的正式接口。这里所涉及的编程接口适用于内核 1.2.13 和 2.0。如果想详细了解 2.1 开发用内核所做的变动，请参见第 17 章“ 最新发展 ”。
正如函数 register_symtab 的名字所暗示，它用来在内核主符号表中注册符号表。这种方法要比通过静态和全局变量的方法清晰的多，这样程序员就可以把关于哪些开放给其他模块，哪些不开放的信息集中存放。这种方法比在源文件中到处堆放 static 声明要好的多。
如果模块在初始化过程中调用了 register_symtab，全局变量就不再是开放的了；只有那些显式罗列在符号表中的符号才开放给内核。
填写一个符号表是项挺复杂的工作，但内核开发人员已经写好了头文件简化这项工作。下面若干行代码演示了如何声明和开放一个符号表：
（代码）
有兴趣的读者可以看看<linux/symtab_begin.h>，但它可是内核中最难懂的头文件之一。事实上，仅想好好使用宏 X 的话，根本没必要读董它。
由于 register_symtab 是在模块加载到内核后被调用的，它可以覆盖模块静态或全局声明的符号。此时，register_symtab 用显式符号表替代模块默认开放的公共符号。
这种覆盖是可能的，因为 insmod 命令处理传递给系统调用 sys_init_module 的全局符号表，然后在调用 init_module 之前注册这个符号表。因此这之后的任何一次显式调用 register_symtab 都会替换相应模块的符号表。
如果你的模块不需要开放任何符号，而且你也不想把所有的东西都声明成 static 的，在 init_module 里加上下面一行语句就可以了。这次对 register_symtab 的调用通过注册一个空表覆盖了模块默认的符号表：
（代码）
如果源文件不想给堆叠在其上的模块提供什么接口，用上面那行语句隐藏所有的符号总是不错的。

当模块从内核卸载时，它所声明的所有公共符号也就自动从主符号表中注销了。不过是全局符号还是显式符号表，这一点都适用。

初始化和终止

正如前面已述，init_module 向内核注册模块所能提供的所有设施。这里我使用了“ 设施 ”，我的意思是指新功能，是一整个设备驱动程序或新软件抽象，是一个可以由应用程序使用的新功能。
通过调用内核函数完成新设施的注册。传递的参数通常为一个指向描述这个新设施的数据结构和要注册的设施名称。这个数据结构通常会包含一些指向模块函数的指针，这就是模块体内的函数是被调用的机制。
除了用来标别模块类别（如字符和块设备驱动程序）的“ 主 ”设施之外，模块还可以注册如下项目：
其他设备

由于这类设施仅仅用于总线型鼠标，这些设备曾一度称为鼠标设备。它们都是些不完整的设备，通常要比那些功能健全的设备简单。

串行端口

可以在运行时向系统里加入串口设备驱动程序；这也是支持 PCMCIA 调治解调器的机制。

行律

行律是处理终端数据流的软件层。模块可以注册新行律，以非标准方式处理终端事务。例如，模块 kmouse 就使用行律从串口鼠标中偷取数据。

终端设备驱动程序

终端设备驱动程序一组实现终端底层数据处理的函数。控制台和串口设备驱动程序为了创建终端设备，它们都要注册自己的驱动程序。而多端口串口则有自己的驱动程序。

/proc 文件

/proc 包含了用来访问内核信息的文件。由于它们也可以用来调试，第 4 章的“ 使用/proc 文件系统 ”将讲解/proc 文件。

二进制文件格式

对于每个可执行文件，内核扫描“ 二进制文件格式 ”列表并按相应的格式执行它。模块可以实现新的格式，Java 模块就是这样做的。

Exec 域

为了提供与其他流行 Unix 系统的兼容，必须修改内核的某些内部表格。一个“ 执行域 ”就是一组从其他操作系统约定到 Linux 系统的映射。例如，模块可以定义执行 SCO 二进制文件的执行域。

符号表

这个已在前面的“ 注册符号表 ”小节中介绍了。

上面这些项目都不是前一章所考虑的设备类型，而且都支持那些通常集成到驱动程序功能中的设施，如/proc 文件和行律。之所以鼠标和其他设备驱动程序都没有象“ 完整 ”字符设备那样管理，这主要是为了方便。过一会儿，当你读到第 3 章“ 字符设备 ”的“ 主从设备号 ”小节时，原因就明了了。
还可以将模块注册为某些驱动程序的附件，但这样做就太特殊了，这里就不作讨论了；它们都使用了“ 注册符号表 ”中讲到的堆叠技术。如果你想做更深一步的探究，你可以在内核源码中查查 register_symtab，并且找找不同驱动程序的入口点。大部分注册函数都是以 register_

开始的，这样你就可以用“ register_ ”在/proc/ksyms 找找它们了。

init_module 中的错误处理

如果你注册时发生什么错误，你必须取消失败前所有已完成的注册。例如，如果系统没有足够内存分配新数据结构时，可能会发生错误。尽管这不太可能，但确实会发生，好的程序代码必须为处理这类事件做好准备。
Linux 不为每个模块保留它都注册了那些设施，因此当 init_module 在某处失败时，模块必须统统收回。如果你在注销你已经注册的设施时失败了，内核就进入一种不稳定状态：卸载模块后，由于它们看起来仍然是“ 忙 ”的，你再也不能注册那些设施了，而且你也无法注销它们了，因为你必须使用你注册时的那个指针，而你不太可能得到那个指针了。恢复这种情况非常复杂，通常，重新启动是最好的解决方法。
我建议你用 goto 语句处理错误恢复。我讨厌使用 goto，但以我个人来看，这是一个它有所做为的地方（而且，是唯一的地方）。在内核里，通常都会象这里处理错误那样使用 goto。
下面这段样例在成功和失败时都能正确执行：
（代码）
返回值（err）是一个错误编码。在 Linux 内核里，错误编码是一个负值，在<linux/errno.h>中定义。如果你不使用其他函数返回的错误编码而要生成自己的，你应该包含<linux/errno.h>，这样就可以使用诸如-ENODEV，-ENOMEM 之类的符号值。总是返回相应的错误编码是种非常好的习惯，因为这样一来用户程序就利用 perror 或相似的方法把它们转换成有意义的字符串了。
很明显，cleanup_module 要取消所有 init_module 中完成的注册。
（代码）

使用计数

为了确定模块是否可以安全地卸载，系统为每个模块保留了一个使用计数。由于模块忙的时候是不能卸载模块的，系统需要这些信息：当文件系统还被安装在系统上时就不能删除这个文件系统类型，而且你也不能在还有程序使用某个字符设备时就去掉它。
通过 3 个宏来维护使用计数：
MOD_INC_USE_COUNT
　　当前模块计数加 1。
MOD_DEC_USE_COUNT
　　计数减 1。
MOD_IN_USE
　　计数非 0 时返回真。
这些宏都定义在<linux/module.h>中，它们都操作不该有程序员直接访问的内部数据结构。事实上，在 2.1 开发版本中，模块管理已经做了非常大的修改，并在 2.1.18 中进行了彻底重写（预知乡情，参见第 17 章的“ 模块化 ”小节）。
注意，cleanup_module 中无需检查 MOD_IN_USE，因为内核在调用清除函数前就已经在系统调用 sys_delete_module（在 kernel/module.c 中定义）中调用完成了检查。
如果忘了更新使用计数，你就不能再卸载模块了。在开发期间这种情况很可能发生，所以你一定要牢记。例如，如果进程因你的驱动程序引用了 NULL 指针而终止，驱动程序就不可

能区关闭设备，使用计数也就无法回复到 0。一种可能的解决方法就是在调试期间完全不使用使用计数，将 MOD_INC_USE_COUNT 和 MOD_DEC_USE_COUNT 重新定义为空操作。另一个解决方法就是利用其他方法将计数强制复位为 0（在第 5 章的“ 使用 ioctl 参数 ”小节中介绍）。在编写成品模块时，决不能投机取巧。然而在调试时期，有时候忽略一些问题可以节省时间，是可以接受的。
使用计数的当前值可以在/proc/modules 中每一项的第 3 个域中找到。这个文件显式系统中当前共加载了那些模块，每一项对应一个模块。其中的域包括，模块名，模块使用的页面数和当前使用计数。这是一个/proc/modules 样例：
（代码）
(autoclean)标志表明模块由 kerneld 管理（见第 11 章）。较新的内核中又加入了一些新的标志，除了一件事外，/proc/modules 的基本结构完全相同：在内核 2.1.18 和更新的版本中，长度用字节计而不是页面计。

卸载

要卸载一个模块就要使用 rmmod 命令。由于无需连编，它的任务远比加载简单。这个命令调用系统调用 delete_module，如果使用计数为 0 它又调用模块的 cleanup_module。
cleanup_module 实现负责注销所有由模块已经注册了的项目。只有符号表是自动删除的。

使用资源

模块不使用资源是无法完成自己的任务的，这些资源包括内存，I/O 端口和中断，如果你要用 DMA 控制器的话，还得有 DMA 通道。
做为一个程序员，你一定已经习惯了内存分配管理，在这方面编写内核代码没什么区别。你的程序使用 kmalloc 分配内存，使用 kfree 释放内存。除了 kmalloc 多一个参数，优先级，外，它们和 malloc，free 很相似。很多情况下，用优先级 GFP_KERNEL 就可以了。缩写 GFP 代表“ Get Free Page（获取空闲页面）。”
与此不同，获取 I/O 端口和中断乍听起来怪怪的，因为程序员一般同用显式的指令访问它们，不必让操作系统了解这些。“ 分配 ”端口和中断与分配内存不同，因为内存是从一个资源池中分配，并且每个地址的行为是一样的；I/O 端口都各有自己的作用，而且驱动程序需要在特定的端口上工作，而不能随便使用某个端口。

端口

对于大多数驱动程序而言，它们的典型工作就是读写端口。不管是初始化还是正常工作的时候，它们都是这样的。为了避免其他驱动程序的干扰，必须保证设备驱动程序以独占方式访问端口——如果一个模块探测因自己的硬件而写某个端口，而恰巧这个端口又是属于另一个设备的，这之后一定会发生点怪事。
为了防止不同设备间的干扰，Linux 的开发者决定实现端口的请求/释放机制。然而，未授权的对端口的访问并不会产生类似于“ 段失效 ”那样的错误——硬件无法支持端口注册。
从文件/proc/ioports 可以以文本方式获得已注册的端口信息，就象下面的样子：
（代码）

文件中的每一项是有驱动程序锁定的范围（以十六进制表示）。在这些被释放前，其他驱动程序不允许访问这些端口。
避免冲突有两个途径。首先，向系统增加新设备的用户检查/proc/ioports，然后在配置新设备使用空闲端口——这种方法假设设备可以通过跳线进行配置。然后，当软件驱动程序初始化自己时，它能自动探测新设备而对其他设备无害：驱动程序不会探测已由其他驱动程序使用的 I/O 端口。
事实上，基于 I/O 注册的冲突避免对于模块化驱动程序很合适，但对于连编到内核里的驱动程序来说却可能失败。尽管我们不涉及这种驱动程序，但还是很是必要注意到，对于一个在启动时初始化自己的驱动程序来说，由于它要使用之后会被注册的端口，很可能会造成对其他设备的误配置。虽然如此，还是没有办法让一个符合规范的驱动程序与已配置好的硬件交互，除非以前加载的驱动程序不注册它的端口。基于以上原因，探测 ISA 设备是件很危险的事，而且如果随正式 Linux 内核发行的驱动程序为了因与尚未加载的模块对应的设备交互，拒绝在模块加载时执行探测功能。
设备探测的问题是因为只有一种方法标别设备，即通过写目标端口然后再读的方法——处理器（而且是任何程序）只能查看数据线上的电子信号。驱动程序编写者知道一旦设备连接到某个特定的端口上，它就会响应相应的查询代码。但是如果另一个设备连到了端口上，程序仍然会写这个设备，但天知道它会怎么响应这个异常的探测操作。有时可以通过读外设的 BIOS，查看一个已知的字串来避免端口探测；已有若干 SCSI 设备使用了这种技术，但并不是每个设备都要有自己的 BIOS。
一个符合规范的驱动程序应该调用 check_region 查看是否某个端口区域已由其他驱动程序锁定，之后就用 request_region 将端口锁住，当驱动程序不再使用端口时调用 release_region 释放端口。这些函数的原型在<linux/ioports.h>中。
注册端口的典型顺序如下所示（函数 skull_probe_hw 包含了所有设备相关代码，这里没有出现）：
（代码）
在 cleanup_module 里释放端口：
（代码）
系统也使用了一套类似的请求/释放策略维护中断，但注册/注销中断比处理端口复杂，整个过程的详细解释将放在第 9 章“ 中断处理 ”中介绍。
与前面讲到的关于设施的注册/注销相似，对资源的请求/释放方法也适合使用已勾勒的基于 goto 的实现框架。
对于编写 PCI 设备驱动程序的人来说，不存在这里所讲的探测问题。我将在第 15 章“ 外部总线简介 ”中介绍。

ISA 内存

本节技术性很强，如果你对处理硬件问题不是很有把握，可以简单跳过这节。
在 Intel 平台上，ISA 槽上的目标设备可能会提供片上内存，范围在 640KB 到 1MB 之间（0xA0000 到 0xFFFFF）；这也是设备驱动程序可以使用的一类资源。
这种内存部件反映了 8086 处理器那个时代，当时 8086 的寻址只有一兆的大小。PC 设计人员决定，低端的 640KB 当做 RAM，而保留另外的 384KB 用于 ROM 和内存映射设备。今天，即便是最强力的个人电脑也还有这个在第一兆字节里的空洞。Linux 的 PC 版保留了这片内存，根本不考虑使用它。本节给出的代码可以让你访问这个区域的内存，但它仅限于 x86 平台，而且 Linux 内核要至少是 2.0.x 的，x 是多少都可以。2.1 版改变了物理内存的访

问方式，比如，640KB-1MB 这段范围内的 I/O 内存就不能再这样访问。访问 I/O 内存的正确方式是第 18 章“ 硬件管理 ”“ 低 1M 内的 ISA 内存 ”小节中的内容，这超出了本章的范围。尽管内核提供了端口和中断的请求/释放机制，当前它还是没能提供给 I/O 内存类似的机制，所以你得自己做了。如果我能理解 Linus 是如何看待 PC 体系结构的化，这里给的方法就不会变化了。

有时某个驱动程序需要在初始化时探测 ISA 内存；例如，我需要告诉视频截取器（frame grabber）在哪映射截取的图象。问题是，如果没有探测方法，我将无法辨别那段范围内哪块内存正在使用。人们需要能够辨别 3 种不同的情况：映射了 RAM，有 ROM（例如，VGA BIOS），或者那段区域空闲。

skull 样例给出一种处理这些内存的方法，但由于 skull 和物理设备无关，它打印完 640KB-1MB 这段内存区域的信息后就退出了。然而，有必要谈一谈用于分析内存的代码，因为它必须处理一些竞争条件。竞争条件就是这样一种情形，两个任务可以竞争同一个资源，而且未同步的操作可能会损坏系统。

尽管驱动程序编写者无需处理多任务，我们还是必须记住，中断可能在你的代码中间发生，而且中断处理函数可能会不提醒你就修改全局量。尽管内核提供了许多工具处理竞争条件，下面给得出的简单规则阐述了处理这个问题的方法；对这个问题的彻底对策将在第 9 章的“ 竞争条件 ”小节中给出。

- 如果仅仅是读取共享的量，而不是写，将其声明为 volatile，要求编译器不对其进行优化。这样，编译好的代码在每次源码读取它时读取这个量了。
- 如果代码需要检查和修改这个值，必须在操作期间关闭中断，这样可以防止其他进程在我们检查过这个值后，但恰恰又在我们修改这个量之前修改这个量。

我们建议采用如下关闭中断的顺序：

（代码）

这里 cli 代表“ clear interrupt flag（清除中断标志）”。上面出现的函数都定义在<asm/system.h>中。

应该避免使用经典的 cli 和 sti 序列，因为有时你无法在关闭中断前断定中断是否打开了。如果此时调用 sti 就是产生很不规则的错误出现，很难追踪这样的错误。

由于那段内存只能通过写物理内存和读取检查才能标别，而且如果测试期间有中断的化，有可能会被其他程序修改，因此检查 RAM 段的代码同时利用了 volatile 声明和 cli。下面的这段代码并不是很简单，如果一个设备正在象它的内存写数据，而这段代码又在扫描这段区域，它就会误认为这段区域是空闲区。好在这样的情况很少发生。

在下面的源代码中，每个 printk 都带有一个 KERN_INFO 前缀。这个符号拼接在格式字串前面做消息的优先级，它定义在<linux/kernel.h>中。这个符号展开后与本章开始的 hello.c 中使用的<1>字串很相似。

（代码）

如果你在探测时注意恢复你所修改的字节，探测内存不会造成与其他设备的冲突。*

作为一个细心的读者，你可能会知道在 15MB-16MB 地址域内的 ISA 内存是怎么回事。很不幸，那是个更棘手的问题，我们将在第 8 章的“ 1M 以上的 ISA 内存 ”小节中讨论。

* 注意，由于某些设备可能将I/O寄存器映射到内存地址上，在某些情况下向内存写数据会带来一些副作用。出于这种考虑以及其他一些考虑，最好不要在产品型驱动程序中使用这里给出的代码。不过它还是可以简单的介绍模块本身，放在这里还是合适的。

自动和手动配置

根据系统的不同，驱动程序需要了解的若干参数也会随之变化。例如，设备必须了解硬件的I/O 地址或内存区域。
注意，本节所讨论的大部分问题并不适用于 PCI 设备（第 15 章介绍）。
根据设备的不同，除了 I/O 地址外，还有一些其他参数会影响系统的驱动程序的行为，如设备的品牌和发行号。驱动程序为了正确地工作有必要了解这些参数的具体值。用正确的数值设置驱动程序（即，配置它）是一项需要在初始化期间完成的复杂的任务。
基本说来，有两种方式可以获得这些正确的数值：或者是用户显式地给出它们，或者是驱动程序自己探测。无疑，自动探测是最好的驱动程序配置方法，而用户配置则是最好实现的；作为驱动程序编写者的一种权衡，他应该尽可能地实现自动配置，但又允许用户配置作为一种可选的方式替代自动配置。这种配置方法的另一个好处就是，在开发期间可以给定参数，从而不用自动探测，可以在以后实现它。
insmod 在加载时接受命令行中给定的整数和字串值，可以给参数赋值。这条命令可以修改在模块中定义的全局变量。例如，如果你的源码中包含了这些变量：
（代码）
那么你就可以使用如下命令加载模块：
（代码）
例子里使用了 printk，它可以显式，当 init_module 被调用时，赋值已经发生了。注意，insmod 可以给任何整型或字符指针变量赋值，不管它们是否是公共符号表中的一部分。但对于声明为数组的串是不能在加载时赋值的，因为它已经在编译时解析出来了，以后就不能修改了。
自动配置可以设计为按如下方式工作：“ 如果配置变量是默认值，就执行自动探测；否则，保留当前值。”为了让这种方法可以工作，“ 默认 ”值应该不是任何用户可以在加载时设定的值。
下面这段代码给出了 skull 是如何自动探测设备的端口地址的。在这个例子中，使用自动探测查找多个设备，而手动配置只限于一个设备。注意，函数 skull_detect 在上面已经给出了，而 skull_init_board 负责完成设备相关的初始化工作，这里没有给出。
（代码）
为了方便用户在 insmod 命令行中给出相应的参数，而且如果这些符号不会放到主符号表中的话，实际使用的驱动程序可以去掉配置变量的前缀（在本例中就是 skull_）。如果它们确实要放到主符号表中，好的办法就是声明两个符号：一个没有前缀，在加载时赋值，一个有前缀，用 register_symtab 放到符号表中。

在用户空间编写驱动程序

到现在为止，一个首次接触内核问题的 Unix 程序员困难会对编写模块非常紧张。写一个用户程序直接读写设备端口可能会更容易些。
事实上，从某些方面看采用用户空间编程更好，而且有时写一个所谓的“ 用户空间设备驱动程序 ”是对内核扩充的明智抉择。
用户空间驱动程序的优点可以总结如下：

- 可以连编完整的 C 库。驱动程序不必寻求许多外部程序的帮助就能维持许多外部任务（实现使用策略的工具程序通常和驱动程序一起发行）。
- 可以用传统的调试器调试驱动程序代码，不必费很大的力气去调试运行中的内核。

- 如果用户空间的驱动程序挂起了，你只要简单地把它杀掉就可以了。驱动程序的问题不太可能将整个系统挂起，除非被控制的硬件真的误操作了。
- 与内核内存不同，用户内存是可以换页的。驱动程序很大但不经常使用的设备除了正在使用时外，不会占用其他程序很多的 RAM。
- 一个细心设计的驱动程序同样可以对设备进行并发访问。

用户空间驱动程序的一个例子就是 X 服务器：它确切地了解它可以操作什么硬件，什么不能，并且给所有的 X 客户提供图形资源。库 libsvga 也是一个类似的程序。
通常，用户空间驱动程序的编写者都实现一个“ 服务器 ”进程，取代内核完成“ 负责硬件控制的唯一代理 ”的任务。客户应用为了最终完成同设备的通信，可以连接这些服务器；一个灵巧的进程可以允许对设备的并发访问。X 服务器就是这样工作的。
用户空间驱动程序的另一个例子是 gpm 鼠标服务器：它完成鼠标设备在不同客户间的仲裁，以便让多个鼠标敏感的应用同时运行在不同的虚终端上。
但有时用户空间驱动程序只授权一个程序访问设备。libsvga 是按这种方式工作的。它连编到应用程序中，在不必依赖集中式设施（如，服务器）的情况就扩展了应用程序的能力。这种方法由于避免了通信代价通常会有更好的性能，但它要求进程以高特权用户的身份运行。
用户空间驱动设备的方法也有很多的缺点。其中最重要的有：

- 在用户空间无法使用中断。除非你学习使用新的 vm86 系统调用而且还要忍受一点性能代价，否则没有方法可以解决这个问题。
- 只能通过 mmap 映射/dev/mem 才能直接访问内存，但只有特权用户才能这样做。
- 只能通过调用 ioperm 或 iopl 才能直接访问 I/O 端口，但只有特权用户才能这样做。
- 由于客户和硬件间传递信息或动作需要一次上下文切换，响应时间很慢。
- 更糟的是，如果驱动程序被换到磁盘后，响应时间会长的不可接受。使用系统调用 mlock 也许会有所助益，但由于用户空间程序要依赖于很多库，一般情况下你都锁住很多页面。
- 很多重要的设备不能在用户空间实现，它们包括网络接口和快设备，但不仅限于此。

正如你所见，用户空间驱动程序毕竟做不了太多的事。但有意义的应用还是存在的：例如，支持 SCSI 扫描仪设备的程序。扫描仪应用程序使用了“ 通用 SCSI ”内核驱动程序，它向用户空间程序开发了低级 SCSI 函数，这样那些程序就可以控制自己的硬件了。
为了写一个用户空间驱动程序，了解一些硬件知识就足够了，而且没有必要去了解内核软件的细节。本书不再对用户级驱动程序进行进一步的讨论，而集中于内核代码。
但另一方面，当处理一些特殊设备时，你可能需要先在用户空间写驱动软件。这样，你在不会挂起整个系统的前提下了解如何控制你的硬件。一旦你完成后，可以很轻松地将这些代码封装到内核模块中。

快速索引

本节总结我们本章所经涉及的内核函数，变量，宏以及/proc 文件。就是说本节可以当做一个参考。如果某项属于某个头文件的话，每一项都会罗列在相关的头文件之后。此后每章的后面都会有类似的一节，总结响应章中出现的新符号。
__KERNEL__
MODULE
预处理符号，编译模块化内核代码时两者都要定义。

```
int init_module(void);
void cleanup_module(void);
```

模块的入口点，必须在模块源码中定义。

#include <linux/module.h>

所需的头文件。模块源码必须包含这个头文件。

MOD_INC_USE_COUNT

MOD_DEC_USE_COUNT

MOD_IN_USE

操作使用计数的宏。

/proc/modules

当前已加载的模块列表。每一项包含模块名，它们占用的内存大小以及使用计数。附加的字串是声明模块当前是否活动的标志。

int register_symtab(struct symbol_table *);

用于指定模块中公共符号的函数。在 2.1.18 以更新的内核中，没有了这个函数。见第 17 章的“ 模块化 ”小节。

int register_symtab_from(struct symbol_table *, long *);

内核 2.0 开发了这个函数而不是 register_symtab，register_symtab 只是一个预处理宏。你可以在/proc/ksyms 中找到 register_symtab_from，但源代码无处与这个函数打交道。

#include <linux/symtab_begin.h>

X(symbol),

#include <linux/symtab_end.h>

用于声明符号表的头文件和预处理宏，用在内核 1.2 和 2.0 中。在 2.1.1 中符号表接口变更了。

#include <linux/version.h>

所需的头文件。除非定义了__NO_VERSION__，否则<linux/module.h>包含了这个头文件。

LINUX_VERSION_CODE

整数宏，对处理版本相关性的#ifdef 很有用。

char kernel_version[] = UTS_RELEASE;

每个模块所需的变量。除非定义了__NO_VERSION__，否则<linux/module.h>定义了这个变量。

__NO_VERSION__

预处理宏。防止<linux/module.h>声明 kernel_version。

#include <linux/sched.h>

最重要的头文件之一。没有它你很难做什么事。

struct task_struct *current;

当前进程。

struct task_struct *current_set[];

Linux 2.0 支持对称多处理，它将 current 定义为一个宏，扩展为 current_set[this_cpu]。你可以在/proc/ksyms 中找到 current_set，但模块代码仍然使用 current。2.1 开发用内核引入一个更快的无需开放内核符号的访问 current 的方式。见第 17 章的“ 其他变化 ”小节。

current->pid

current->comm

当前进程的进程 ID 和命令名。

#include <linux/kernel.h>

int printk(const char *fmt, …);

在内核代码中使用的与 printf 相似的函数。

#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);

在内核代码中使用的与 malloc 和 free 相似的函数。优先级一般采用 GFP_KERNEL。

#include <linux/ioport.h>
int check_region(unsigned int from, unsigned int extent);
void reguest_region(unsigned int from, unsigned int extent, const char *name);
void release_region(unsigned int from, unsigned int extent);

用于申请和释放 I/O 端口的函数。版本 2.1.30 将 unsigned int 参数变为 unsigned long，但这个变化不会影响驱动程序代码。

#include <asm/system.h>

定义诸如 save_flags 和 restore_flags 之类访问机器寄存器的宏的头文件。

save_flags(long flags);
restore_flags(long flags);

预处理宏，可以允许临时修改处理标志。

cli();
sti();

关闭和打开中断。不应该使用 sti；而要使用 save_flags 和 restore_flags。

/proc/ksyms

公共内核符号表。

/proc/ioports

已安装设备的端口列表。

第 3 章 字符设备驱动程序

本章的目标是编写一个完整的字符设备驱动程序。由于这类驱动程序适合于大多数简单的硬件设备，我们首先开放一个字符设备驱动程序。字符也相对比较好理解，比如说块设备驱动程序。我们的最终目标是写一个模块化的字符设备驱动程序，但本章我们不再讲述有关模块化的问题。

本章通篇都是从一个真实的设备驱动程序截取出的代码块：这个设备就是 scull，是“ Simple Character Utility for Loading Localities ”的缩写。尽管 scull 是一个设备，但它却是操作内存的字符设备。这种情况的一个副作用就是，只要涉及 scull，“ 设备 ”这个词就可以同“ scull 使用的内存区 ”互换使用。

scull 的优点是，由于每台电脑都有内存，所以它与硬件无关。scull 用 kmalloc 分配内存，而且仅仅操作内存。任何人都可以编译和运行 scull，而且 scull 可以移植到所有 Linux 支持的平台上。但另一方面，除了演示内核于字符设备驱动程序间的交互过程，可以让用户运行某些测试例程外，scull 做不了“ 有用的 ”事。

scull 的设计

编写设备驱动程序的第一步就是定义驱动程序提供给用户程序的能力（“ 机制 ”）。由于我们的“ 设备 ”是电脑内存的一部分，我做什么都可以。它可以是顺便存取设备，也可以是随机存取设备，可以是一个设备，也可以是多个，等等。

为了是 scull 更有用，可以成为编写真实设备的驱动程序的模板，我将向你展示如何在电脑的内存之上实现若干设备抽象操作，每一种操作都有自己的特点。

scull 的源码实现如下设备。由模块实现的每一种设备都涉及一种类型：

scull0-3

4 个设备，共保护了 4 片内存区，都是全局性的和持久性的。“ 全局性 ”是指，如果打开设备多次，所有打开它的文件描述符共享其中的数据。“ 持久性 ”是指，如果设备关闭后再次打开，数据不丢失。由于可以使用常用命令访问这个设备，如 cp，cat 以及 shell I/O 重定向等，这个设备操作非常有趣；本章将深入探讨它的内部结构。

scullpipe0-3

4 个“ fifo ”设备，操作起来有点象管道。一个进程读取另一个进程写入的数据。如果有多个进程读同一个设备，他们彼此间竞争数据。通过 scullpipe 的内部结构可以了解阻塞型和非阻塞型读/写是如何实现的；没有中断也会出现这样的情况。尽管真实的驱动程序利用中断与它们的设备同步，但阻塞型和非阻塞型操作是非常重要的内容，从概念上讲与中断处理（第 9 章，中断处理，介绍）无关。

scullsingle

scullpriv

sculluid

scullwuid

这些设备与 scull0 相似，但在何时允许 open 操作时都不同方式的限制。第一个（scullsingle）只允许一次一个进程使用驱动程序，而 scullpriv 对每个虚拟控制台是私有的（每个设备对虚拟控制台是私有的）。sculluid 和 scullwuid 可以多次打开，但每次只能有一个用户；如果另

一个用户锁住了设备，前者返回-EBUSY，而后者则实现为阻塞型 open。通过这些可以展示如何实现不同的访问策略。
每一个 scull 设备都展示了驱动程序不同的功能，而且都不同的难度。本章主要讲解 scull0-3 的内部结构；第 5 章，字符设备驱动程序的扩展操作，将介绍更复杂的设备： “ 一个样例实现：scullpipe ” 介绍 scullpipe，“ 设备文件的访问控制 ” 介绍其他设备。

主设备号和次设备号

通过访问文件系统的名字（或 “ 节点 ”）访问字符设备，通常这些文件位于/dev 目录。设备文件是特殊文件，这一点可以通过 ls -l 输出的第一列中的“c”标明，它说明它们是字符节点。/dev 下还有块设备，但它们的第一列是“ b ”;尽管如下介绍的某些内容也同样适用于块设备，现在我们只关注字符设备。如果你执行 ls 命令，在设备文件条目的最新修改日期前你会看到两个数（用逗号分隔），这个位置通常显示文件长度。这些数就是相应设备的主设备号和次设备号。下面的列表给出了我使用的系统上的一些设备。它们的主设备号是 10，1 和 4，而次设备号是 0，3，5，64-65 和 128-129。
（代码）
主设备号标识设备对应的驱动程序。例如，/dev/null 和/dev/zero 都有驱动程序 1 管理，而所有的 tty 和 pty 都由驱动程序 4 管理。内核利用主设备号将设备与相应的驱动程序对应起来。次设备号只由设备驱动程序使用；内核的其他部分不使用它，仅将它传递给驱动程序。一个驱动程序控制若干个设备并不为奇（如上面的例子所示）——次顺便号提供了一种区分它们的方法。
向系统增加一个驱动程序意味着要赋予它一个主设备号。这一赋值过程应该在驱动程序（模块）的初始化过程中完成，它调用如下函数，这个函数定义在<linux/fs.h>：
（代码）
返回值是错误码。当出错时返回一个负值；成功时返回零或正值。参数 major 是所请求的主设备号，name 是你的设备的名字，它将在/proc/devices 中出现，fops 是一个指向跳转表的指针，利用这个跳转表完成对设备函数的调用，本章稍后将在 “ 文件操作 ” 一节中介绍这些函数。
主设备号是一个用来索引静态字符设备数组的整数。在 1.2.13 和早期的 2.x 内核中，这个数组有 64 项，而 2.0.6 到 2.1.11 的内核则升至 128。由于只有设备才处理次设备号，register_chrdev 不传递次设备号。
一旦设备已经注册到内核表中，无论何时操作与你的设备驱动程序的主设备号匹配的设备文件，内核都会通过在 fops 跳转表索引调用驱动程序中的正确函数。
接下来的问题就是如何给程序一个它们可以请求你的设备驱动程序的名字。这个名字必须插入到/dev 目录中，并与你的驱动程序的主设备号和次设备号相连。
在文件系统上创建一个设备节点的命令是 mknod，而且你必须是超级用户才能创建设备。除了要创建的节点名字外，该命令还带三个参数。例如，命令：
（代码）
创建一个字符设备（c），主设备号是 127，次设备号是 0。由于历史原因，次设备号应该在 0-255 范围内，有时它们存储在一个字节中。存在很多原因扩展可使用的次设备号的范围，但就现在而言，仍然有 8 位限制。

动态分配主设备号

某些主设备号已经静态地分配给了大部分公用设备。在内核源码树的Documentation/device.txt文件中可以找到这些设备的列表。由于许多数字已经分配了，为新设备选择一个唯一的号码是很困难的——不同的设备要不主设备号多得多。
很幸运（或是感谢某些人天才），你可以动态分配主设备号了。如果你调用 register_chrdev 时的 major 为零的话，这个函数就会选择一个空闲号码并做为返回值返回。主设备号总是正的，因此不会和错误码混淆。
我强烈推荐你不要随便选择一个一个当前不用的设备号做为主设备号，而使用动态分配机制获取你的主设备号。
动态分配的缺点是，由于分配给你的主设备号不能保证总是一样的，无法事先创建设备节点。然而这不是什么问题，这是因为一旦分配了设备号，你就可以从/proc/devices 读到。为了加载一个设备驱动程序，对 insmod 的调用被替换为一个简单的脚本，它通过/proc/devices 获得新分配的主设备号，并创建节点。
/proc/devices 一般如下所示：
（代码）
加载动态分配主设备号驱动程序的脚本可以利用象 awk 这类工具从/proc/devices 中获取信息，并在/dev 中创建文件。
下面这个脚本，scull_load，是 scull 发行中的一部分。使用以模块形式发行的驱动程序的用户可以在/etc/rc.d/rc.local 中调用这个脚本，或是在需要模块时手工调用。此外还有另一种方法：使用 kerneld。这个方法和其他模块的高级功能将在第 11 章“ Kerneld 和高级模块化 ”中介绍。
（代码）
这个脚本同样可以适用于其他驱动程序，只要重新定义变量和调整 mknod 那几行就可以了。上面那个脚本创建 4 个设备，4 是 scull 源码中的默认值。
脚本的最后两行看起来有点怪怪的：为什么要改变设备的组和权限呢？原因是这样的，由 root 创建的节点自然也属于 root。默认权限位只允许 root 对其有写访问权，而其他只有读权限。正常情况下，设备节点需要不同的策略，因此需要进行某些修改。通常允许一组用户访问对设备，但实现细节却依赖于设备和系统管理员。安全是个大问题，这超出了本书的范围。scull_load 中的 chmod 和 chgrp 那两行仅仅是最为处理权限问题的一点提示。稍后，在第 5 章的“ 设备文件的访问控制 ”一节中将介绍 sculluid 源码，展示设备驱动程序如何实现自己的设备访问授权。
如果重复地创建和删除/dev 节点似乎有点过分的话，有一个解决的方法。如果你看了内核源码 fs/devices.c 的话，你可以看到动态设备号是从 127（或 63）之后开始的，你可以用 127 做为主设备号创建一个长命节点，同时可以避免在每次相关设备加载时调用脚本。如果你使用了几个动态设备，或是新版本的内核改变了动态分配的特性，这个技巧就不能用了。（如果内核发生了修改，基于内核内部结构编写的代码并不能保证继续可以工作。）不管怎样，由于开发期间模块要不断地加载和卸载，你会发现这一技术在开发期间还是很有用的。
就我看来，分配主设备号的最佳方式是，默认采用动态分配，同时留给你在加载时，甚至是编译时，指定主设备号的余地。使用我建议的代码将与自动端口探测的代码十分相似。scull 的实现使用了一个全局变量，scull_major，来保存所选择的设备号。该变量的默认值是 SCULL_MAJOR，在所发行的源码中为 0，即“ 选择动态分配 ”。用户可以使用这个默认值或选择某个特定的主设备号，既可以在编译前修改宏定义，也可以在 ins_mod 命令行中指定。

最后，通过使用 scull_load 脚本，用户可以在 scull_load 中命令行中将参数传递给 insmod。这里是我在 scull.c 中使用的获取主设备号的代码：
（代码）

从系统中删除设备驱动程序

当从系统中卸载一个模块时，应该释放主设备号。这一操作可以在 cleanup_module 中调用如下函数完成：
（代码）
参数是要释放的主设备号和相应的设备名。内核对这个名字和设备号对应的名字进行比较：如果不同，返回-ENINVAL。如果主设备号超出了所允许的范围或是并未分配给这个设备，内核一样返回-EINVAL。在cleanup_module中注销资源失败会有非常不号的后果。下次读取/proc/devices时，由于其中一个name字串仍然指向模块内存，而那片内存已经不存在了，系统将产生一次失效。这种失效称为Oops*，内核在访问无效地址时将打印这样的消息。
当你卸载驱动程序而又无法注销主设备号时，这种情况是无法恢复的，即便为此专门写一个“补救”模块也无济于事，因为 unregister_chrdev 中调用了 strcmp，而 strcmp 将使用未映射的 name 字串，当释放设备时就会使系统 Oops。无需说明，任何视图打开这个异常的设备号对应的设备的操作都会 Oops。
除了卸载模块，你还经常需要在卸载驱动程序时删除设备节点。如果设备节点是在加载时创建的，可以写一个简单的脚本在卸载时删除它们。对于我们的样例设备，脚本 scull_unload 完成这个工作。如果动态节点没有从/dev 中删除，就会有可能造成不可预期的错误：如果动态分配的主设备号相同，开发者计算机上的一个空闲/dev/framegrabber 就有可能在一个月后引用一个火警设备。“没有这个文件或目录”要比这个新设备所产生的后果要好得多。

dev_t 和 kdev_t

到目前为止，我们已经谈论了主设备号。现在是讨论次设备号和驱动程序如何使用次设备号来区分设备的时候了。
每次内核调用一个设备驱动程序时，它都告诉驱动程序它正在操作哪个设备。主设备号和次设备号合在一起构成一个数据类型并用来标别某个设备。设备号的组合（主设备号和次设备号合在一起）驻留在稍后介绍的“inode”结构的 i_rdev 域中。每个驱动程序接收一个指向 struct inode 的指针做为第一个参数。这个指针通常也称为 inode，函数可以通过查看 inode->i_rdev 分解出设备号。
历史上，Unix 使用 dev_t 保存设备号。dev_t 通常是<sys/types.h>中定义的一个 16 位整数。而现在有时需要超过 256 个次设备号，但是由于有许多应用（包括 C 库在内）都了解 dev_t 的内部结构，改变 dev_t 是很困难的，如果改变 dev_t 的内部结构就会造成这些应用无法运行。因此，dev_t 类型一直没有改变；它仍是一个 16 位整数，而且次设备号仍限制在 0-255 内。然而，在 Linux 内核内部却使用了一个新类型，kdev_t。对于每一个内核函数来说，这个新类型被设计为一个黑箱。它的想法是让用户程序不能了解 kdev_t。如果 kdev_t 一直是隐藏的，它可以在内核的不同版本间任意变化，而不必修改每个人的设备驱动程序。
有关 kdev_t 的信息被禁闭在<linux/kdev_t.h>中，其中大部分是注释。如果你对代码后的哲

* 怪里怪气的Linux爱好者将“Oops”这个词既当成名词也当成动词使用。

学感兴趣的话，这个头文件是一段很有指导性的代码。因为<linux/fs.h>已经包含了这个头文件，没有必要显式地包含这个文件。
不幸的是，kdev_t 类型是一个“现代”概念，在内核版本 1.2 中没有这个类型。在较新的内核中，所有的引用设备的内核变量和结构字段都是 kdev_t 的，但是在 1.2.13 中同样的变量却是 dev_t 的。如果你的驱动程序只使用它接收的结构字段，而不声明自己的变量的话，这不会有什么问题的。如果你需要声明自己的设备类型变量，为了可移植性你应该在你的头文件中加入如下几行：
（代码）
这段代码是样例源码中的 sysdep.h 头文件的一部分。我不会在源码中在引用 dev_t，但是要假设前一个条件语句已经执行了。
如下这些宏和函数是你可以对 kdev_t 执行的操作：
MAJOR(kdev_t dev);
从 kdev_t 结构中分解出主设备号。
MINOR(kdev_t dev);
分解出次设备号。
MKDEV(int ma, int mi);
通过主设备号和次设备号返回 kdev_t。
kdev_t_to_nr(kdev_t dev);
将 kdev_t 转换为一个整数（dev_t）。
to_kdev_t(int dev);
将一个整数转换为 kdev_t。注意，核心态中没有定义 dev_t，因此使用了 int。
与 Linux 1.2 相关的头文件定义了同样的操作 dev_t 的函数，但没有那两个转换函数，这也就是为什么上面那个条件代码简单地将它们定义返回它们的参数值。

文件操作

在接下来的几节中，我们将看看驱动程序能够对它管理的设备能够完成哪些不同的操作。在内核内部用一个 file 结构标别设备，而且内核使用 file_operations 结构访问驱动程序的函数。这一设计是我们所看到的 Linux 内核面向对象设计的第一个例证。我们将在以后看到更多的面向对象设计的例证。file_operations 结构是一个定义在<linux/fs.h>中的数指针表。结构 struct file 将在以后介绍。
我们已经 register_chrdev 调用中有一个参数是 fops，它是一个指向一组操作（open，read 等等）表的指针。这个表的每一个项都指向由驱动程序定义的处理相应请求的函数。对于你不支持的操作，该表可以包含 NULL 指针。对于不同函数的 NULL 指针，内核具体的处理行为是不同的，下一节将逐一介绍。
随着新功能不断加入内核，file_operations结构已逐渐变得越来越大（尽管从 1.2.0 到 2.0.x并没有增加新字段）。这种增长应该不会有什么副作用，因为在发现任何尺寸不匹配时，C编译器会将全局或静态struct变量中的未初始化字段填 0。新的字段都加到结构的末尾*，所以在编译时会插入一个NULL指针，系统会选择默认行为（记住，对于所有模块需要加载的新内核，都要重新编译一次模块）。
在 2.1 开发用内核中，有些与 fops 字段相关的函数原型发生了变化。这些变化将在第 17 章“近期发展”的“文件操作”一节中介绍。

* 例如，版本 2.1.31 增加一个称为lock的新字段。

纵览不同操作

下面的列表将介绍应用程序能够对设备调用的所有操作。这些操作通常称为“方法”，用面向对象的编程术语来说就是说明一个对象声明可以操作在自身的动作。
为了使这张列表可以用来当作索引，我尽量使它简洁，仅仅介绍每个操作的梗概以及当使用NULL时的内核默认行为。你可以在初次阅读时跳过这张列表，以后再来查阅。
在介绍完另一个重要数据结构（file）后，本章的其余部分将讲解最重要的一些操作并提供一些提示，告诫和真实的代码样例。由于我们尚不能深入探讨内存管理和异步触发机制，我们将在以后的章节中介绍这些更为复杂操作。
struct file_operations 中的操作按如下顺序出现，除非注明，它们的返回0时表示成功，发生错误时返回一个负的错误编码：
int (*lseek)(struct inode *, struct file *, off_t, int);
方法 lseek 用来修改一个文件的当前读写位置，并将新位置做为（正的）返回值返回。出错时返回一个负的返回值。如果驱动程序没有设置这个函数，相对与文件尾的定位操作失败，其他定位操作修改 file 结构（在“file 结构”中介绍）中的位置计数器，并成功返回。2.1.0 中该函数的原型发生了变化，第 17 章“原型变化”将讲解这些内容。
int (*read)(struct inode *, struct file *, char *, int);
用来从设备中读取数据。当其为 NULL 指针时将引起 read 系统调用返回-EINVAL（“非法参数”）。函数返回一个非负值表示成功的读取了多少字节。
int (*write)(struct inode *, struct file *, const char *, int);
向设备发送数据。如果没有这个函数，write 系统调用向调用程序返回一个-EINVAL。注意，版本 1.2 的头文件中没有 const 这个说明符。如果你自己在 write 方法中加入了 const，当与旧头文件编译时会产生一个警告。如果你没有包含 const，新版本的内核也会产生一个警告；在这两种情况你都可以简单地忽略这些警告。如果返回值非负，它就表示成功地写入的字节数。
int (*readdir)(struct inode *, struct file *, void *, filldir_t);
对于设备节点来说，这个字段应该为 NULL；它仅用于目录。
int (*select)(struct inode *, struct file *, int, select_table *);
select 一般用于程序询问设备是否可读和可写，或是否一个“异常”条件发生了。如果指针为 NULL，系统假设设备总是可读和可写的，而且没有异常需要处理。“异常”的具体含义是和设备相关的。在当前的 2.1 开发用内核中，select 的实现方法完全不同。（见第 17 章的“poll 方法”）。返回值告诉系统条件满足（1）或不满足（0）。
int (*ioctl)(struct inode *, struct file *, unsigned int, unsigned long);
系统调用 ioctl 提供一中调用设备相关命令的方法（如软盘的格式化一个磁道，这既不是读操作也不是写操作）。另外，内核还识别一部分 ioctl 命令，而不必调用 fops 表中的 ioctl。如果设备不提供 ioctl 入口点，对于任何内核没有定义的请求，ioctl 系统调用将返回-EINVAL。当调用成功时，返回给调用程序一个非负返回值。
int (*mmap)(struct inode *, struct file *, struct vm_area_struct *);
mmap 用来将设备内存映射到进程内存中。如果设备不支持这个方法，mmap 系统调用将返回-ENODEV。
int (*open)(struct inode *, struct file *);
尽管这总是操作在设备节点上的第一个操作，然而并不要求驱动程序一定要声明这个方法。如果该项为 NULL，设备的打开操作永远成功，但系统不会通知你的驱动程序。
void (*release)(struct inode *, struct file *);

当节点被关闭时调用这个操作。与 open 相仿，release 也可以没有。在 2.0 和更早的核心中，close 系统调用从不失败；这种情况在版本 2.1.31 中有所变化（见第 17 章）。

int (*fsync)(struct inode *, struct file *);

刷新设备。如果驱动程序不支持，fsync 系统调用返回-EINVAL。

int (*fasync)(struct inode *, struct file *, int);

这个操作用来通知设备它的 FASYNC 标志的变化。异步触发是比较先进的话题，将在第 5 章的“异步触发”一节中介绍。如果设备不支持异步触发，该字段可以是 NULL。

int (*check_media_change)(kdev_t dev);

check_media_change 只用于块设备，尤其是象软盘这类可移动介质。内核调用这个方法判断设备中的物理介质（如软盘）自最近一次操作以来发生了变化（返回 1）或是没有（0）。字符设备无需实现这个函数。

int (*revalidate)(kdev_t dev);

这是最后一项，与前面提到的那个方法一样，也只适用于块设备。revalidate 与缓冲区高速缓存有关。我们将在第 12 章“加载块设备驱动程序”的“可移动设备”中介绍 revalidate。

scull 驱动程序中适用的 file_operations 结构如下：

（代码）

在最新的开发用内核中，某些原型已经发生了变化。该列表是从 2.0.x 的头文件中提炼出来的，这里给出的原型对于大多数内核而言都是正确的。内核 2.1 引入的变化（以及为了使我们的模块可移植所进行的修改）在针对不同操作的每一节和第 17 章的“文件操作”中详细介绍。

file 结构

在<linux/fs.h>中定义的 struct file 是设备驱动程序所适用的又一个最重要的数据结构。注意，file 与用户程序中的 FILE 没有任何关联。FILE 是在 C 库中定义且从不出现在内核代码中。而 struct file 是一个内核结构，从不出现在用户程序中。

file 结构代表一个“打开的文件”。它有内核在 open 时创建而且在 close 前做为参数传递给如何操作在设备上的函数。在文件关闭后，内核释放这个数据结构。一个“打开的文件”与由 struct inode 表示的“磁盘文件”有所不同。

在内核源码中，指向 struct file 的指针通常称为 file 或 filp（“文件指针”）。为了与这个结构相混淆，我将一直称指针为 filp－flip 是一个指针（同样，它也是设备方法的参数之一），而 file 是结构本身。

struct file 中的最重要的字段罗列如下。与上节相似，这张列表在首次阅读时可以略过。在下一节中，我们将看到一些真正的 C 代码，我将讨论某些字段，到时你可以反过来查阅这张列表。

mode_t f_mode;

文件模式由 FMODE_READ 和 FMODE_WRITE 标别。你可能需要在你的 ioctl 函数中查看这个域来来检查读/写权限，但由于内核在调用你的驱动程序的 read 和 write 前已经检查了权限，你无需检查在这两个方法中检查权限。例如，一个不允许的写操作在驱动程序还不知道的情况下就被已经内核拒绝了。

loff_t f_ops;

当然读/写位置。loff_t 是一个 64 位数值（用 gcc 的术语就是 long long）。如果驱动程序需要知道这个值，可以直接读取这个字段。如果定义了 lseek 方法，应该更新 f_pos 的值。当传输数据时，read 和 write 也应该更新这个值。

unsigned short f_flags;
文件标志，如 O_RDONLY，O_NONBLOCK 和 O_SYNC。驱动程序为了支持非阻塞型操作需要检查这个标志，而其他标志很少用到。注意，检查读/写权限应该查看 f_mode 而不是 f_flags。所有这些标志都定义在<linux/fcntl.h>中。
struct inode *f_inode;
打开文件所对应的 i 节点。inode 指针是内核传递给所有文件操作的第一个参数，所以你一般不需要访问 file 结构的这个字段。在某些特殊情况下你只能访问 struct file 时，你可以通过这个字段找到相应的 i 节点。
struct file_operations *f_op;
与文件对应的操作。内核在完成 open 时对这个指针赋值，以后需要分派操作时就读这些数据。filp->f_op 中的值从不保存供以后引用；这也就是说你可以在需要的事后修改你的文件所对应的操作，下一次再操作那个打开文件的相应操作时就会调用新方法。例如，主设备号为 1 的设备（/dev/null，/dev/zero 等等）的 open 代码根据要打开的次设备号替换 filp->f_op 中的操作。这种技巧有助于在不增加系统调用负担的情况下方便识别主设备号相同的设备。能够替换文件操作的能力在面向对象编程技术中称为 “ 方法重载 ”。
void *private_data;
系统调用 open 在调用驱动程序的 open 方法前将这个指针置为 NULL。驱动程序可以将这个字段用于任意目的或者忽略简单忽略这个字段。驱动程序可以用这个字段指向已分配的数据，但是一定要在内核释放 file 结构前的 release 方法中清除它。private_data 是跨系统调用保存状态信息的非常有用的资源，在我们的大部分样例都使用了这个资源。
实际的结构里还有其他一些字段，但它们对于驱动程序并不是特别有用。由于驱动程序从不填写 file 结构；它们只是简单地访问别处创建的结构，我们可以大胆地忽略这些字段。

Open 和 Close

现在让我们已经走马观花地看了一遍这些字段，下面我们将开始在实际的 scull 函数中使用这些字段。

Open 方法

open 方法是驱动程序用来为以后的操作完成初始化准备工作的。此外，open 还会增加设备计数，以便防止文件在关闭前模块被卸载出内核。
在大部分驱动程序中，open 完成如下工作：

- 检查设备相关错误（诸如设备未就绪或相似的硬件问题）。
- 如果是首次打开，初始化设备。
- 标别次设备号，如有必要更新 f_op 指针。
- 分配和填写要放在 filp->private_data 里的数据结构。
- 增加使用计数。

在 scull 中，上面的大部分操作都要依赖于被打开设备的次设备号。因此，首先要做的事就是标别要操作的是哪个设备。我们可以通过查看 inode->i_rdev 完成。
我们已经谈到内核是如何不使用次设备号的了，因此驱动程序可以随意使用次设备号。事实上，利用不同的次设备号访问不同的设备，或以不同的方式打开同一个设备。例如，/dev/ttyS0 和/dev/ttyS1 是两个不同的串口，而/dev/cua0 的物理设备与/dev/ttyS0 相同，仅仅是操作行为

不同。cua是“ 调出 ”设备 ；它们不是终端，而且它们也没有终端所需要的所有软件支持 (即，它们没有加入行律*)。所有的串口设备都有许多不同的次设备号，这样驱动程序就区分它们了：ttyS与cua不一样。

驱动程序从来都不知道被打开的设备的名字，它仅仅知道设备号——而且用户可以按照自己的规范给用设备起别名，而完全不用原有的名字。如果你看看/dev 目录就会知道，你将发现对应相同主/次设备号的不同名字；设备只有一个而且是相同的，而且没有方法区分它们。例如，在许多系统中，/dev/psaux 和/dev/bmouseps2 都存在，而且它们有同样的设备号；它们可以互换使用。后者是 “ 历史遗迹 ”，你的系统里可以没有。

scull驱动程序是这样使用次设备号的：最高 4 位标别设备类型个体（personality），如果该类型可以支持多实例（scull0-3 和scullpipe0-3），低 4 位可以供你标别这些设备。因此，scull0 的高 4 位与scullpipe0 不同，而scull0 的低 4 位与scull1 不同*。源码中定义了两个宏（TYPE 和NUM）从设备号中分解出这些位，我们马上就看到这些宏。

对于每一设备类型，scull 定义了一个相关的 file_operations 结构，并在 open 时替换 filp->f_op。下面的代码就是位切分和多 fops 是如何实现的：

（代码）

内核根据主设备号调用 open；scull 用上面给出的宏处理次设备号。接着用 TYPE 索引 scull_fop_array 数组，从中分解出被打开设备的方法集。

我在 scull 中所做的就是根据次设备号的类型给 filp->f_op 赋上正确的值。然后调用新的 fops 中定义的 open 方法。通常，驱动程序不必调用自己的 fops，它只有内核分配正确的驱动程序方法时调用。但当你的 open 方法不得不处理不同设备类型时，在根据被打开设备次设备号修改 fops 指针后就需要调用 fops->open 了。

scull_open 的实际代码如下。它使用了前面那段代码中定义的 TYPE 和 NUM 两个宏来切分次设备号：

（代码）

这里给一点解释。用来保存内存区的数据结构是 Scull_Dev，这里简要介绍一下。Scull_Dev 和 scull_trim （“ Scull 的内存使用 ” 一节中讨论）的内部结构这里并没有使用。全局变量 scull_nr_devs 和 scull_devices[] (全部小写) 分别是可用设备数和指向 Scull_Dev 的指针数组。

这段代码看起来工作很少，这是因为当调用 open 时它没做任何针对某个设备的处理。由于 scull0-3 设备被设计为全局的和永久性的，这段代码无需做什么。特别是，由于我们无法维护 scull 的打开计数，也就是模块的使用计数，因此没有类似于 “ 首次打开时初始化设备 ” 这类动作。

唯一实际操作在设备上的操作是，当设备写打开时将设备截断为长度 0。截断是 scull 设计的一部分：用一个较短的文件覆盖设备，以便缩小设备数据区，这与普通文件写打开截断为 0 很相似。

但 “ 打开是截断 ” 有一个严重的缺点：若干因为某些原因设备内存正在使用，释放这些内存会导致系统失效。尽管可能性不大，这种情况总会发生：如果 read 和 write 方法在数据传输时睡眠了，另一个进程可能写打开这个设备，这时麻烦就来了。处理竞争条件是一个相当高级的主题，我将在第 9 章的 “ 竞争条件 ” 中讲解。scull 处理这个问题的简单方法就是在内存还是使用时不释放内存，“ Scull 的内存使用 ” 一节中将说明。

以后我们看到其他 scull 个体（personality）时将会看到一个真正的初始化工作如何完成。

---

* “ 行律 ” 是用来处理终端I/O策略的软件模块。

* 位切分一种典型的使用次设备号的方式。例如，IDE驱动程序就使用高 2 位表示磁盘号，低 6 位表示分区号。

release 方法

release 方法的作用正好与 open 相反。这个设备方法有时也称为 close。它应该：

- 使用计数减 1。
- 释放 open 分配在 filp->private_data 中的内存。
- 在最后一次关闭操作时关闭设备。

scull的基本模型无需进行关闭设备动作，所以所需代码是很少的*：

（代码）

使用计数减 1 是非常重要的，因为如果使用计数不归 0，内核是不会卸载模块的。

如果某个时刻一个从没被打开的文件被关闭了计数将如何保证一致呢？我们都知道，dup 和 fork 都会在不调用 open 的情况下，将一个打开文件复制为 2 个，但每一个都会在程序终止时关闭。例如，大多数程序从来不打开它们的 stdin 文件（或设备），但它们都会在终止关闭它。

答案很简单。如果 open 没有调用，release 也不会调。内核维护一个 file 结构被使用了多少次的使用计数。无论是 fork 还是 dup 都不创建新的数据结构；它们仅是增加已有结构的计数。

新的 struct file 仅由 open 创建。只有在该结构的计数归 0 时 close 系统调用才会执行 close 方法，这只有在删除这个结构时才会进行。close 方法与 close 系统调用间的关系保证了模块使用计数永远是一致的。

Scull 的内存使用

在介绍读写操作以前，我们最好先看看 scull 如何完成内存分配以及为什么要完成内存分配。为了全面理解代码我们需要知道“如何分配”，而“为什么”则反映了驱动程序编写者需要做出的选择，尽管 scull 绝不是一个典型设备，但同样需要。

本节只讲解 scull 中的内存分配策略，而不会讲解你写实际驱动程序时需要的硬件管理技巧。这些技巧将在第 8 章“硬件管理”和第 9 章中介绍。因此，如果你对针对内存操作的 scull 驱动程序的内部工作原理不感兴趣的话，你可以跳过这一节。

scull 使用的内存，这里也称为“设备”，是变长的。你写的越多，它就增长得越多；消减的过程只在用短文件覆盖设备时发生。

所选的实现 scull 的方法不是很聪明。实现较聪明的源码会更难读，而且本节的目的只是讲解 read 和 write，而不是内存管理。这也就是为什么虽然整个页面分配会更有效，但代码只使用了 kmalloc 和 kfree，而没有涉及整个页面的分配的操作。

而另一面，从理论和实际角度考虑，我又不想限制“设备”区的尺寸。理论上将，给所管理的数据项强加任何限制总是很糟糕的想法。从实际出发，为了测试系统在内存短缺时的性能，scull 可以帮助将系统的剩余内存用光。进行这样的测试有助于你理解系统的内部行为。你可以使用命令 cp /dev/zero /dev/scull 用光所有的物理内存，而且你也可以用工具 dd 选择复制到 scull 设备中多少数据。

在 scull 中，每个设备都是一组指针的链表，而每一个指针又指向一个 Scull_Dev 结构。每一个这样的结构通过一个中间级指针数组最多可引用 4,000,000 个字节。发行的源码中使用

* 由于scull_open用不同的fops替换了filp->f_ops，不同种类的设备会使用不同的函数完成关闭操作，这一点我们将在后面看到。

了一个有 1000 个指针的数组，每个指针指向 4000 个字节。我把每一个内存区称为一个“ 量子 ”，数组（或它的长度）称为“ 量子集 ”。scull 设备和它的内存区如图 3-1 所示。
所选择的数字是这样的，向 scull 写一个字节就会消耗内存 8000 了字节：每个量子 4 个，量子集 4 个（在大多数平台上，一个指针是 4 个字节；当在 Alpha 平台编译时量子集本身就会耗费 8000 个字节，在 Alpha 平台上指针是 8 个字节）。但另一方面，如果你向 scull 写大量的数据，由于每 4MB 数据只对应一个表项，而且设备的最大尺寸只限于若干 MB，不可能超出计算机内存的大小，遍历这张链表的代价不是很大。
为量子和量子集选择合适的数值是一个策略问题，而非机制问题，而且最优数值依赖于如何使用设备。源码中为处理这些问题允许用户修改这些值：

- 在编译时，可以修改 scull.h 中的 SCULL_QUANTUM 和 SCULL_QSET。
- 在加载时，可以利用 insmod 修改 scull_quantum 和 scull_qset 整数值。
- 在运行时，用 ioctl 方法改变默认值和当前值。ioctl 将在第 5 章的“ ioctl ”一节中介绍。

使用宏和整数值进行编译时和加载时配置让人想起前面提到的如何选择主设备号。无论何时驱动程序需要一个随意的数值或这个数值与策略相关，我都使用这种技术。
留下来的唯一问题就是如何选择默认数值。尽管有时驱动程序编写者也需要事先调整配置参数，但他们在编写自己的模块时不会碰到同样的问题。在这个特殊的例子里，问题的关键在于寻找因未填满的量子和量子集导致的内存浪费和量子和量子集太小带来的分配、释放和指针连接等操作的代价之间的平衡。
此外，还必须考虑 kmalloc 的内部设计。现在我们还无法讲述太多的细节，只能简单规定“ 比 2 次幂稍小一点是最佳尺寸 ”比较好。kmalloc 的内部结构将在第 7 章“ Getting Hold of Memory ”的“ The Real Story of kmalloc ”一节中探讨。
默认数值的选择基于这样的假设，大部分程序员不会受限与 4MB 的物理内存，那样大的数据量有可能会写到 scull 中。一台内存很多的计算机的属主可能因测试向设备写数十 MB 的数据。因此，所选的默认值是为了优化中等规模的系统和大数据量的使用。
保存设备信息的数据机构如下：
（代码）
下面的代码给出了实际工作时是如何利用 Scull_Dev 保存数据的。其中给出的函数负责释放整个数据区，并且在文件写打开时由 scull_open 调用。如果当前设备内存正在使用，该函数就不释放这些内存（象“ open 方法 ”中所说那样）；否则，它简单地遍历链表，释放所有找到的量子和量子集。
（代码）

读和写

读写 scull 设备也就意味着要完成内核空间和用户进程空间的数据传输。由于指针只能在当前地址空间操作，而驱动程序运行在内核空间，数据缓冲区则在用户空间，这一操作不能通过通常利用指针或 memcpy 完成。
由于驱动程序不过怎样都要在内核空间和用户缓冲区间复制数据，如果目标设备不是 RAM 而是扩展卡，也有同样的问题。事实上，设备驱动程序的主要作用就是管理设备（内核空间）和应用（用户空间）间的数据传输。
在 Linux 里，跨空间复制是通过定义在<asm/segment.h>里的特殊函数实现的。完成这种操作函数针对不同数据尺寸（char，short，int，long）进行了优化；它们中的大部分将在第 5 章的“ 使用 ioctl 参数 ”一节中介绍。
scull 中 read 和 write 的驱动程序代码需要完成到用户空间和来自用户空间的整个数据段的复

制。下面这些提供这些功能，它们可以传输任意字节：
（代码）
这两个函数的名字可以追溯到第 1 版 Linux，那时唯一支持的体系结构是 i386，而且 C 代码中还可以窥见许多汇编码。在 Intel 平台上，Linux 通过 FS 段寄存器访问用户空间，到 Linux 2.0 时仍沿用了以前的名字。在 Linux 2.1 中它们改变了，但是 2.0 是本书的主要目标。详情可见第 17 章的“ 访问用户空间 ”。
尽管上面介绍的函数看起来很象正常的 memcpy 函数，但当在内核代码中访问用户空间时必须额外注意一些问题；正在被访问的用户页面现在可能不在内存中，而且页面失效处理函数有可能在传送页面的时候让进程进入睡眠状态。例如，必须从交换区读取页面时会发生这种情况。对驱动程序编写者来说，静效果就是对于任何访问用户空间的函数都必须是可重入的，而且能够与其他驱动程序函数并发执行。这就是为什么 scull 实现中不允许在 dev->usage 不为 0 时释放设备：read 和 write 方法在它们使用 memcpy 函数前先完成 usage 计数加 1。
现在谈谈实际的设备方法，读方法的任务是将数据从设备复制到用户空间（使用 memcpy_tofs），而写方法必须将数据从用户空间复制到设备（使用 memcpy_tofs）。每一个 read 或 write 系统调用请求传输一定量的字节，但驱动程序可以随意传送其中一部分数据——读与写的具体规则稍有不同。
如果有错误发生，read 和 write 都返回一个负值。返回给调用程序一个大于等于 0 的数值，告诉它成功传输了多少字节。如果某个数据成功地传输了，随后发生了错误，返回值必须是成功传输的字节计数，而错误直到下次函数被调用时才返回给程序。
read 的不同参数的相应功能如图 3-2 所示。
（图 3-2）
内核函数返回一个负值通知错误，该数的数值表示已经发生的错误种类（如第 2 章“ 编写和运行模块 ”的“ init_module 中的错误处理 ”所述），运行在用户空间的程序访问变量 errno 获知发生什么错误。这两方面的不同行为，一方面坏是靠库规范强加的，另一方面是内核不处理 errno 的优点导致的。
现在再来谈谈可移植性，你一定会注意到 read 和 write 方法中参数 count 的类型总是 int，但在内核 2.1.0 中却变为 unsigned long。而且，由于 long 既可以表示 count 也可以表示负的错误编码，方法的返回值也从 int 变为了 long。
类型的修改是有益的：对于 count 来说，unsigned long 与 int 相比是一个更好的选择，它有更宽广的值域。这种选择非常好，Alpha 小组在 2.1 尚为发行时就修改了类型（主要是因为 GNU C 库在它们的系统调用定义中使用的是 unsigned long）。
尽管有好处，这一修改却带来了驱动程序代码的平台相关性。为了绕过这个问题，所有的 O’Reilly FTP 站点上的样例模块都使用了如下定义（来自 sysdep.h）：
（代码）
在宏计算后，read 和 write 的 count 计数就永远声明为 count_t，而返回值则是 read_write_t 的。我为没有使用 typedef，而是使用了宏定义，这是因为加入 typedef 后会引起更多的编译告警（见第 10 章“ 明智使用数据类型 ”中的“ 接口相关类型 ”一节）。另一方面，函数原型中出现大写的名字很难看，所以我使用标准 typedef 规范命名新“ 类型 ”。
第 17 章将更详细地介绍版本 2.1 的可移植性。

read 方法

调用程序对 read 返回值的解释如下：

- 如果返回值等于最为 count 参数传递给 read 系统调用的值，所请求的字节数传输就成功

完成了。这是最好的情况。

- 如果返回值是正的，但是比 count 小，只有部分数据成功传送。这种情况因设备的不同可能有许多原因。大部分情况下，程序会重新读数据。例如，如果你用 fread 函数读数据，这个库库函数会不断调用系统调用直至所请求的数据传输完成。
- 如果返回值为 0，它表示已经到达了文件尾。
- 负值意味着发生了错误。值就是错误编码，错误编码在<linux/errno.h>中定义。

上面的表格中遗漏了一种情况，就是“ 没有数据，但以后会有 ”。在这种情况下，read 系统调用应该阻塞。我们将在第 5 章的“ 阻塞型 I/O ”一节中处理阻塞出入。

scull 代码利用了这些规则。特别是，它利用了部分读规则。每一次调用 scull_read 只处理一个数据量子，而不必实现循环收集所有数据；这样一来代码就更短更易读了。如果读程序确实需要更多的数据，它可以重新调用这个调用。如果用标准库读取设备，应用程序都不会注意到数据传送的量子化过程。

如果当前的读位置超出了设备尺寸，scull 的 read 方法就返回 0 告知程序这里已经没有数据了（换句话说就是，我们已经到文件尾了）。如果进程 A 正在读设备，而此时进程 B 写打开这个设备，于是将设备截断为长度 0，这种情况就发生了。进程 A 突然发现自己超过了文件尾，并且在下次调用 read 时返回 0。

这里是 read 的代码：

（代码）

write 方法

与 read 相似，根据如下返回值规则，write 也可以传输少于请求的数据量：

- 如果返回值等于 count，则完成了请求数目的字节传送。
- 如果返回值是正的，但小于 count，只传输了部分数据。再说明一次，程序很可能会再次读取余下的部分。
- 如果值为 0，什么也没写。这个结果不是错误，而且也没有什么缘由需要返回一个错误编码。再说明一次，标准库会重复调用 write。以后的章节会介绍阻塞型 write，我们会对这种情形最更详尽的考察。
- 负值意味发生了错误；语义与 read 相同。

很不幸，有些错误程序在发生部分传输时会报错并异常退出。最显著的例子就是一个不算旧版本的 GNU 文件工具，它就有这样的错误。如果你的按照是 1995 年或更早的（例如，Slackware 2.3），你的 cp 在处理 scull 会失败。如果在 cp 写一块比一个量子大的数据时，你见到这样一条消息/dev/scull: no such file or directory，你就是用的这个版本的 cp。GNU dd 最初就被设计为拒绝读写部分块，而 cat 拒绝写部分块。因此，不应该用 cat 访问 scull，而 dd 则应该传递与 scull 的量子大小相同的块。注意，注意，这一缺陷在 scull 的实现中可以弥补，但我不想把代码搞的太复杂，能说明问题就行了。

与 scull 的 read 代码相同，write 代码每次只处理一个量子：

（代码）

试试新设备

一旦你了解了刚刚讲解的 4 个方法，驱动程序就可以编译和测试了；它保留你写的数据，直至你用新覆盖它们。这个设备有点象长度只受物理 RAM 容量限制的数据缓冲区。你可以试

试 cp，dd 以及输入/输出重定向等命令和机制来测试这个驱动程序。
根据你向 scull 里写了多少数据，用 free 命令可以看到空闲内存的缩减和扩增。
为了进一步证实读写确实是每次一个量子的，你可以在驱动程序的适当位置增加 printk，通过它来看看在程序读或写大数据块到底系统是如何工作的。另外，还可以用工具 strace 来监视程序调用的系统调用，以及它们的返回值。跟踪 cp 或 ls -l > /dev/scull0 会显示出量子化的读写。下一章将详细介绍监视（或跟踪）技术。

快速索引

本章介绍了下列符合头文件。struct file_operations 和 struct file 的字段列表这里没有给出。
#include <linux/fs.h>
“ 文件系统 ”头文件，它是编写设备驱动程序必须的头文件。所有重要的函数都在这里声明。
int register_chrdev(unsigned int major, const char *name, struct file_operations *fops);
注册字符设备驱动程序。如果主设备不为 0，它就不加修改的使用；如果主设备号为 0，系统动态给这个设备赋一个新设备号。
int unregister_chrdev(unsigned int major, const char *name);
在卸载时注销驱动程序。major 和 name 字串都必须存放与注册驱动程序时相同的值。
kdev_t inode->i_rdev;
通过传递给每个设备方法的 inode 参数可以访问的当前设备的设备“ 号 ”。
int MAJOR(kdev_t dev);
int MINOR(kdev_t dev);
这两个宏从设备项中分解出主次设备号。
kdev_t MKDEV(int major, int minor);
这个宏由主次设备号构造 kdev_t。
#include <asm/segment.h>
2.0 及以上内核定义跨空间复制函数的头文件。这些函数是那些从用户段到内核段复制数据或反之的函数。版本 2.1 修改了这些函数以及头文件的名字（预知详情可见第 17 章的“ 访问用户空间 ”）。
void memcpy_fromfs(void *to, const void *from, unsigned long count);
void memcpy_tofs(void *to, const void *from, unsigned long count);
这些函数用来从用户空间到内核空间或反之复制字节数组。“ FS ”是用来在内核代码中访问用户空间的 i386 段寄存器。这些函数在版本 2.1 修改了。

第 4 章 调试技术

对于任何编写内核代码的人来说，最吸引他们注意的问题之一就是如何完成调试。由于内核是一个不与某个进程相关的功能集，其代码不能很轻松地放在调试器中执行，而且也不能跟踪。
本章介绍你可以用来监视内核代码和跟踪错误的技术。

用打印信息调试

最一般的调试技术就是监视，就是在应用内部合适的点加上 printf 调用。当你调试内核代码的时候，你可以用 printk 完成这个任务。

Printk

在前些章中，我们简单假设 printk 工作起来和 printf 很类似。现在是介绍一下它们之间不同的时候了。
其中一个不同点就是，printk 允许你根据它们的严重程度，通过附加不同的“记录级”来对消息分类，或赋予消息优先级。你可以用宏来指示记录级。例如，KERN_INFO，我们前面已经看到它被加在打印语句的前面，它就是一种可能的消息记录级。记录级宏展开为一个字串，在编译时和消息文本拼接在一起；这也就是为什么下面的例子中优先级和格式字串间没有逗号。这有两个 printk 的例子，一个是调试信息，一个是关键信息：
（代码）
在<linux/kernel.h>中定义了 8 种记录级别串。没有指定优先级的 printk 语句默认使用 DEFAULT_MESSAGE_LOGLEVEL 优先级，它是一个在 kernel/printk.c 中定义的整数。默认记录级的具体数值在 Linux 的开发期间曾变化过若干次，所以我建议你最好总是指定一个合适的记录级。
根据记录级，内核将消息打印到当前文本控制台上：如果优先级低于 console_loglevel 这个数值的话，该消息就显示在控制台上。如果系统同时运行了 klogd 和 syslogd，无论 console_loglevel 为何值，内核都将消息追加到/var/log/messages 中。
变量 console_loglevel 最初初始化为 DEFAULT_CONSOLE_LOGLEVEL，但可以通过 sys_syslog 系统调用修改。如 klogd 的手册所示，可以在启动 klogd 时指定-c 开关来修改这个变量。此外，你还可以写个程序来改变控制台记录级。你可以在 O’Reilly 站点上的源文件中找到我写的一个这种功能的程序，miscprogs/setlevel.c。新优先级是通过一个 1 到 8 之间的整数值指定的。
你也许需要在内核失效后降低记录级（见“调试系统故障”），这是因为失效处理代码会将 console_loglevel 提升到 15，之后所有的消息都会出现在控制台上。为看到你的调试信息，如果你运行的是内核 2.0.x 话，你需要提升记录级。内核 2.0 发行降低了 MINIMUM_CONSOLE_LOGLEVEL，而旧版本的 klogd 默认情况下要打印很多控制消息。如果你碰巧使用了这个旧版本的守护进程，除非你提升记录级，内核 2.0 会比你预期的打印出更少的消息。这就是为什么 hello.c 中使用了<1>标记，这样可以保证消息显示在控制台上。

从 1.3.43 一来的内核版本通过允许你向指定虚控制台发送消息，藉此提供一个灵活的记录策略。默认情况下，“ 控制台 ”是当前虚终端。也可以选择不同的虚终端接收消息，你只需向所选的虚终端调用 ioctl(TIOCLINUX)。如下程序，setconsole，可以用来选择哪个虚终端接收内核消息；它必须以超级用户身份运行。如果你对 ioctl 还不有把握，你可以跳过这至下一节，等到读完第 5 章“ 字符设备驱动程序的扩展操作 ”的“ ioctl ”一节后，再回到这里读这段代码。

（代码）

setconsole 使用了用于 Linux 专用功能的特殊的 ioctl 命令 TIOCLINUX。为了使用 TIOCLINUX，你要传递给它一个指向字节数组的指针。数组的第一个字节是所请求的子命令的编码，随后的字节依命令而不同。在 setconsole 中使用了子命令 11，后一个字节（存放在 bytes[1]中）标别虚拟控制台。TIOCLINUX 的完成介绍可以在内核源码 drivers/char/tty_io.c 中找到。

消息是如何记录的

printk 函数将消息写到一个长度为 LOG_BUF_LEN 个字节的循环缓冲区中。然后唤醒任何等待消息的进程，即那些在调用 syslog 系统调用或读取/proc/kmesg 过程中睡眠的进程。这两个访问记录引擎的接口是等价的。不过/proc/kmesg 文件更象一个 FIFO 文件，从中读取数据更容易些。一跳简单的 cat 命令就可以读取消息。

如果循环缓冲区填满了，printk 就绕到缓冲区的开始处填写新数据，覆盖旧数据。于是记录进程就丢失了最旧的数据。这个问题与利用循环缓冲区所获得的好处相比可以忽略不计。例如，循环缓冲区可以使系统在没有记录进程的情况下照样运行，同时又不浪费内存。Linux 处理消息的方法的另一个特点是，可以在任何地方调用 printk，甚至在中断处理函数里也可以调用，而且对数据量的大小没有限制。这个方法的唯一缺点就是可能丢失某些数据。

如果 klogd 正在运行，它读取内核消息并将它们分派到 syslogd，它随后检查/etc/syslog.conf 找到处理这些数据的方式。syslogd 根据一个“ 设施 ”和“ 优先级 ”切分消息；可以使用的值定义在<sys/syslog.h>中。内核消息根据相应 printk 中指定的优先级记录到 LOG_KERN 设施中。如果 klogd 没有运行，数据将保存在循环缓冲区中直到有进程来读取数据或数据溢出。

如果你不希望因监视你的驱动程序的消息而把你的系统记录搞乱，你给klogd指定-f（文件）选项或修改/etc/syslog.conf将记录写到另一个文件中。另一种方法是一种强硬方法：杀掉klogd，将消息打印到不用的虚终端上*，或者在一个不用的xterm上执行cat /proc/kmesg显示消息。

使用预处理方便监视处理

在驱动程序开发早期，printk 可以对调试和测试新代码都非常有帮助。然而当你正式发行驱动程序时，你应该去掉，或者至少关闭，这些打印语句。很不幸，你可能很快就发现，随着你想不再需要那些消息并去掉它们时，你可能又要加新功能，你又需要这些消息了。解决这些问题有几种方法——如何从全局打开和关闭消息以及如何打开和关闭个别消息。

下面给出了我处理消息所用的大部分代码，它有如下一些功能：

- 可以通过在宏名字加一个字母或去掉一个字母打开或关闭每一条语句。

* 例如，使用命令setlevel 8；set console 10 设置终端 10 显示消息。

- 通过在编译前修改 CFLAGS 变量，可以一次关闭所有消息。
- 同样的打印语句既可以用在内核态（驱动程序）也可以用在用户态（演示或测试程序）。

下面这些直接来自 scull.h 的代码片断实现了这些功能。
（代码）
符合 PDEBUG 和 PDEBUGG 依赖于是否定义了 SCULL_DEBUG，它们都和 printf 调用很类似。
为了进一步方便这个过程，在你的 Makefile 加上如下几行。
（代码）
本节所给出的代码依赖于 gcc 对 ANSI C 预编译器的扩展，gcc 可以支持带可变数目参数的宏。这种对 gcc 的依赖并不是什么问题，因为内核对 gcc 特性的依赖更强。此外，Makefile 依赖于 GNU 的 gmake；基于同样的道理，这也不是什么问题。
如果你很熟悉 C 预编译器，你可以将上面的定义扩展为可以支持“调试级”概念的，可以为每级赋一个整数（或位图），说明这一级打印多么琐碎的消息。
但是每一个驱动程序都有它自己的功能和监视需求。好的编程技巧会在灵活性和高效之间找到一个权衡点，这个我就不能说哪个对你最好了。记住，预编译器条件（还有代码中的常量表达式）只到编译时运行，你必须重新编译程序来打开或关闭消息。另一种方法就是使用 C 条件语句，它在运行时运行，因此可以让你在程序执行期间打开或关闭消息。这个功能很好，但每次代码执行系统都要进行额外的处理，甚至在消息关闭后仍然会影响性能。有时这种性能损失是无法接受的。
个人观点，尽管上面给出的宏迫使你每次要增加或去掉消息时都要重新编译，重新加载模块，但我觉得用这些宏已经很好了。

通过查询调试

上一节谈到了 printk 是如何工作的以及如何使用它。但没有谈及它的缺点。
由于 syslogd 会一直保持刷新它的输出文件，每打印一行都会引起一次磁盘操作，因此过量使用 printk 会严重降低系统性能。至少从 syslogd 的角度看是这样的。它会将所有的数据都一股脑地写到磁盘上，以防在打印消息后系统崩溃；然而，你不想因为调试信息的缘故而降低系统性能。这个问题可以通过在/etc/syslogd.conf 中记录文件的名字前加一个波折号解决，但有时你不想修改你的配置文件。如果不这样，你还可以运行一个非 klogd 的程序（如前面介绍的 cat /proc/kmesg），但这样并不能为正常操作提供一个合适的环境。
与这相比，最好的方法就是在你需要信息的时候，通过查询系统获得相关信息，而不是持续不断地产生数据。事实上，每一个 Unix 系统都提供了很多工具用来获得系统信息：ps，netstat，vmstat 等等。
有许多技术适合与驱动程序开发人员查询系统，简而言之就是，在/proc 下创建文件和使用 ioctl 驱动程序方法。

使用/proc 文件系统

Linux 中的/proc 文件系统与任何设备都没有关系——/proc 中的文件都在被读取时有核心创建的。这些文件都是普通的文本文件，它们基本上可由普通人理解，也可被工具程序理解。例如，对于大多数 Linux 的 ps 实现而言，它都通过读取/proc 文件系统获得进程表信息的。/proc 虚拟文件的创意已由若干现代操作系统使用，且非常成功。

/proc 的当前实现可以动态创建 i 节点，允许用户模块为方便信息检索创建如何入口点。
为了在/proc 中创建一个健全的文件节点（可以 read，write，seek 等等），你需要定义 file_operations 结构和 inode_operations 结构，后者与前者有类似的作用和尺寸。创建这样一个 i 节点比起创建整个字符设备并没有什么不同。我们这里不讨论这个问题，如果你感兴趣，你可以在源码树 fs/proc 中获得进一步细节。
与大多数/proc 文件一样，如果文件节点仅仅用来读，创建它们是比较容易的，我将这里介绍这一技术。很不幸，这一技术只能在 Linux 2.0 及其后续版本中使用。
这里是创建一个称为/proc/scullmem 文件的 scull 代码，这个文件用来获取 scull 使用的内存信息。
（代码）
填写/proc 文件非常容易。你的函数获取一个空闲页面填写数据；它将数据写进缓冲区并返回所写数据的长度。其他事情都由/proc 文件系统处理。唯一的限制就是所写的数据不能超过 PAGE_SIZE 个字节（宏 PAGE_SIZE 定义在头文件<asm/page.h>中；它是与体系结构相关的，但你至少可以它有 4KB 大小）。
如果你需要写多于一个页面的数据，你必须实现功能健全的文件。
注意，如果一个正在读你的/proc文件的进程发出了若干read调用，每一个都获取新数据，尽管只有少量数据被读取，你的驱动程序每次都要重写整个缓冲区。这些额外的工作会使系统性能下降，而且如果文件产生的数据与下一次的不同，以后的read调用要重新装配不相关的部分，这一会造成数据错位。事实上，由于每个使用C库的应用程序都大块地读取数据，性能并不是什么问题。然而，由于错位时有发生，它倒是一个值得考虑的问题。在获取数据后，库调用至少要调用 1 次read——只有当read返回 0 时才报告文件尾。如果驱动程序碰巧比前面产生了更多的数据，系统就返回到用户空间额外的字节并且与前面的数据块是错位的。我们将在第 6 章“ 时间流 ”的“ 任务队列 ”一节中涉及/proc/jiq*，那时我们还会遇到错位问题。
cleanup_module 中应该使用下面的语句注销/proc 节点：
（代码）
传递给函数的参数是包含要撤销文件的目录名和文件的 i 节点号。由于 i 节点号是自动分配的，在编译时是无法知道的，必须从数据结构中读取。

ioctl 方法

ioctl，下一章将详细讨论，是一个系统调用，它可以操做在文件描述符上；它接收一个“ 命令 ”号和（可选的）一个参数，通常这是一个指针。
做为替代/proc 文件系统的方法，你可以为调试实现若干 ioctl 命令。这些命令从驱动程序空间复制相关数据到进程空间，在进程空间里检查这些数据。
只有使用 ioctl 获取信息比起/proc 来要困难一些，因为你一个程序调用 ioctl 并显示结果。必须编写这样的程序，还要编译，保持与你测试的模块间的一致性等。
不过有时候这是最好的获取信息的方法，因为它比起读/proc 来要快得多。如果在数据写到屏幕前必须完成某些处理工作，以二进制获取数据要比读取文本文件有效得多。此外，ioctl 不限制返回数据的大小。
ioctl 方法的一个优点是，当调试关闭后调试命令仍然可以保留在驱动程序中。/proc 文件对任何查看这个目录的人都是可见的，然而与/proc 文件不同，未公开的 ioctl 命令通常都不会被注意到。此外，如果驱动程序有什么异常，它们仍然可以用来调试。唯一的缺点就是模块

*

会稍微大一些。

通过监视调试

有时你遇到的问题并不特别糟，通过在用户空间运行应用程序来查看驱动程序与系统之间的交互过程可以帮助你捕捉到一些小问题，并可以验证驱动程序确实工作正常。例如，看到 scull 的 read 实现如何处理不同数据量的 read 请求后，我对 scull 更有信心。
有许多方法监视一个用户态程序的工作情况。你可以用调试器一步步跟踪它的函数，插入打印语句，或者用 strace 运行程序。在实际目的是查看内核代码时，最后一项技术非常有用。strace 命令是一个功能非常强大的工具，它可以现实程序所调用的所有系统调用。它不仅可以显示调用，而且还能显示调用的参数，以符号方式显示返回值。当系统调用失败时，错误的符号值（如，ENOMEM）和对应的字串（Out of memory）同时显示。strace 还有许多命令行选项；最常用的是-t，它用来显示调用发生的时间，-T，显示调用所花费的时间，以及-o，将输出重定向到一个文件中。默认情况下，strace 将所有跟踪信息打印到 stderr 上。
strace 从内核接收信息。这意味着一个程序无论是否按调试方式编译（用 gcc 的-g 选项）或是被去掉了符号信息都可以被跟踪。与调试器可以连接到一个运行进程并控制它类似，你还可以跟踪一个已经运行的进程。
跟踪信息通常用来生成错误报告报告给应用开发人员，但是对内核编程人员来说也一样非常有用。我们可以看到系统调用是如何执行驱动程序代码的；strace 允许我们检查每一次调用输入输出的一致性。
例如，下面的屏幕输出给出了命令 ls /dev > /dev/scull0 的最后几行：
（代码）
很明显，在 ls 完成目标目录的检索后首次对 write 的调用中，它试图写 4KB。很奇怪，只写了 4000 个字节，接着重试这一操作。然而，我们知道 scull 的 write 实现每次只写一个量子，我在这里看到了部分写。经过若干步骤之后，所有的东西都清空了，程序正常退出。
另一个例子，让我们来读 scull 设备：
（代码）
正如所料，read 每次只能读到 4000 个字节，但是数据总量是不变的。注意本例中重试工作是如何组织的，注意它与上面写跟踪的对比。wc 专门为快速读数据进行了优化，它绕过了标准库，以便每次用一个系统调用读取更多的数据。你可以从跟踪的 read 行中看到 wc 每次要读 16KB。
Unix 专家可以在 strace 的输出中找到很多有用信息。如果你被这些符号搞得满头雾水，我可以只看文件方法（open，read 等等）是如何工作的。
个人认为，跟踪工具在查明系统调用的运行时错误过程中最有用。通常应用或演示程序中的 perror 调用不足以用来调试，而且对于查明到底是什么样的参数触发了系统调用的错误也很有帮助。

调试系统故障

即便你用了所有监视和调试技术，有时候驱动程序中依然有错误，当这样的驱动程序执行会会造成系统故障。当这种情况发生时，获取足够多的信息来解决问题是至关重要的。
注意，“ 故障 ”不意味着“ panic ”。Linux 代码非常鲁棒，可以很好地响应大部分错误：故障通常会导致当前进程的终止，但系统继续运行。如果在进程上下文之外发生故障，或是组成

系统的重要部件发生故障时，系统可能 panic。但问题出在驱动程序时，通常只会导致产生故障的进程终止——即那个使用驱动程序的进程。唯一不可恢复的损失就是当进程被终止时，进程上下文分配的内存丢失了；例如，由驱动程序通过 kmalloc 分配的动态链表可能丢失。然而，由于内核会对尚是打开的设备调用 close，你的驱动程序可以释放任何有 open 方法分配的资源。
我们已经说过，当内核行为异常时会在控制台上显示一些有用的信息。下一节将解释如何解码和使用这些消息。尽管它们对于初学者来说相当晦涩，处理器的给出数据都是些很有意思的信息，通常无需额外测试就可以查明程序错误。

Oops 消息

大部分错误都是 NULL 指针引用或使用其他不正确的指针数值。这些错误通常会导致一个 oops 消息。
由处理器使用的地址都是“ 虚 ”地址，而且通过一个复杂的称为页表（见第 13 章“ Mmap 和 DMA ”中的“ 页表 ”一节）的结构映射为物理地址。当引用一个非法指针时，页面映射机制就不能将地址映射到物理地址，并且处理器向操作系统发出一个“ 页面失效 ”。如果地址确实是非法的，内核就无法从失效地址上“ 换页 ”；如果此时处理在超级用户太，系统于是就产生一个“ oops ”。值得注意的是，在版本 2.1 中内核处理失效的方式有所变化，它可以处理在超级用户态的非法地址引用了。新实现将在第 17 章“ 最近发展 ”的“ 处理内核空间失效 ”中介绍。
oops 显示故障时的处理器状态，模块 CPU 寄存器内容，页描述符表的位置，以及其他似乎不能理解的信息。这些是由失效处理函数（arch/*/kernel/traps.c）中的 printk 语句产生的，而且象前面“ Printk ”一节介绍的那样进行分派。
让我们看看这样一个消息。这里给出的是传统个人电脑（x86 平台），运行 Linux 2.0 或更新版本的 oops——版本 1.2 的输出稍有不同。
（代码）
上面的消息是在一个有意加入错误的失效模块上运行 cat 所至。fault.c 崩溃如下代码：
（代码）
由于 read 从它的小缓冲区（faulty_buf）复制数据到用户空间，我们希望读一小块文件能够工作。然而，每次读出多于 1KB 的数据会跨越页面边界，如果访问了非法页面 read 就会失败。事实上，前面给出的 oops 是在请求一个 4KB 大小的 read 时发生的，这条消息在 /var/log/messages（syslogd 默认存放内核消息的文件）的 oops 消息前给出了：
（代码）
同样的 cat 命令却不能在 Alpha 上产生 oops，这是因为从 faulty_buf 读取 4KB 字节没有超出页边界（Alpha 上的页面大小是 8KB，缓冲区正好在页面的起始位置附近）。如果在你的系统上读取 faulty 没有产生 oops，试试 wc，或者给 dd 显式地指定块大小。

使用 ksymoops

oops 消息的最大问题就是十六进制数值对于程序员来说没什么意义；需要将它们解析为符号。
内核源码通过其所包含的 ksymoops 工具帮助开发人员——但是注意，版本 1.2 的源码中没有这个程序。该工具将 oops 消息中的数值地址解析为内核符号，但只限于 PC 机产生的 oops

消息。由于消息本身就是处理器相关的，每一体系结构都有其自身的消息格式。
ksymoops 从标准输入获得 oops 消息，并从命令行内核符号表的名字。符号表通常就是/usr/src/linux/System.map。程序以更可读的方式打印调用轨迹和程序代码，而不是最原始的 oops 消息。下面的片断就是用上一节的 oops 消息得出的结果：
（代码）
由 ksymoops 反汇编出的代码给出了失效的指令和其后的指令。很明显——对于那些知道一点汇编的人——repz movsl 指令（REPeat till cx is Zero, MOVe a String of Longs）用源索引（esi，是 0x202e000）访问了一个未映射页面。用来获得模块信息的 ksymoops -m 命令给出，模块映射到一个在 0x0202dxxx 的页面上，这也确认乐 esi 确实超出了范围。
由于 faulty 模块所占用的内存不在系统表中，被解码的调用轨迹还给出了两个数值地址。这些值可以手动补充，或是通过 ksyms 命令的输出，或是在/proc/ksyms 中查询模块的名字。然而对于这个失效，这两个地址并不对应与代码地址。如果你看了 arch/i386/kernel/traps.c，你就发现，调用轨迹是从整个堆栈并利用一些启发式方法区分数据值（本地变量和函数参数）和返回地址获得的。调用轨迹中只给出了引用内核代码的地址和引用模块的地址。由于模块所占页面既有代码也有数据，错综复杂的栈可能会漏掉启发式信息，这就是上面两个 0x202xxxx 地址的情况。
如果你不愿手动查看模块地址，下面这组管道可以用来创建一个既有内核又有模块符号的符号表。无论何时你加载模块，你都必须重新创建这个符号表。
（代码）
这个管道将完整的系统表与/proc/ksyms 中的公开内核符号混合在一起，后者除了内核符号外，还包括了当前内核里的模块符号。这些地址在 insmod 重定位代码后就出现在/proc/ksyms 中。由于这两个文件的格式不同，使用了 sed 和 awk 将所有的文本行转换为一种合适的格式。然后对这张表排序，去除重复部分，这样 ksymoops 就可以用了。
如果我们重新运行 ksymoops，它从新的符号表中截取出如下信息：
（代码）
正如你所见到的，当跟踪与模块有关的 oops 消息时，创建一个修订的系统表是很有助益的：现在 ksymoops 能够对指令指针解码并完成整个调用轨迹了。还要注意，显式反汇编码的格式和 objdump 所使用的格式一样。objdump 也是一个功能强大的工具；如果你需要查看失败前的指令，你调用命令 objdump –d faulty.o。
在文件的汇编列表中，字串 faulty_read+45/60 标记为失效行。有关 objdump 的更多的信息和它的命令行选项可以参见该命令的手册。
即便你构建了你自己的修订版符号表，上面提到的有关调用轨迹的问题仍然存在：虽然 0x202xxxx 指针被解码了，但仍然是假的。
学会解码 oops 消息需要一定的经验，但是确实值得一做。用来学习的时间很快就会有所回报。不过由于机器指令的 Unix 语法与 Intel 语法不同，唯一的问题在于从哪获得有关汇编语言的文档；尽管你了解 PC 汇编语言，但你的经验都是用 Intel 语法的编程获得的。在参考书目中，我给一些有所补益的书籍。

使用 oops

使用 ksymoops 有些繁琐。你需要 C++编译器编译它，你还要构建你自己的符号表来充分发挥程序的能力，你还要将原始消息和 ksymoops 输出合在一起组成可用的信息。
如果你不想找这么多麻烦，你可以使用 oops 程序。oops 在本书的 O’Reilly FTP 站点给出的源码中。它源自最初的 ksymoops 工具，现在它的作者已经不维护这个工具了。oops 是用 C

语言写成的，而且直接查看/proc/ksyms 而无需用户每次加载模块后构建新的符号表。
该程序试图解码所有的处理器寄存器并堆栈轨迹解析为符号值。它的缺点是，它要比 ksymoops 罗嗦些，但通常你所有的信息越多，你发现错误也就越快。oops 的另一个优点是，它可以解析 x86，Alpha 和 Sparc 的 oops 消息。与内核源码相同，这个程序也按 GPL 发行。
oops 产生的输出与 ksymoops 的类似，但是更完全。这里给出前一个 oops 输出的开始部分—由于在这个 oops 消息中堆栈没保存什么有用的东西，我不认为应该显示整个堆栈轨迹：
（代码）
当你调试“ 真正的 ”模块（faulty 太短了，没有什么意义）时，将寄存器和堆栈解码是非常有益的，而且如果被调试的所有模块符号都开放出来时更有帮助。在失效时，处理器寄存器一般不会指向模块的符号，只有当符号表开放给/proc/ksyms 时，你才能输出中标别它们。
我们可以用一下步骤制作一张更完整的符号表。首先，我们不应在模块中声明静态变量，否则我们就无法用 insmod 开放它们了。第二，如下面的截取自 scull 的 init_module 函数的代码所示，我们可以用#ifdef SCULL_DEBUG 或类似的宏屏蔽 register_symtab 调用。
（代码）
我们在第 2 章“ 编写和运行模块 ”的“ 注册符号表 ”一节中已经看到了类似内容，那里说，如果模块不注册符号表，所有的全局符号就都开放。尽管这一功能仅在 SCULL_DEBUG 被激活时才有效，为了避免内核中的名字空间污染，所有的全局符号有合适的前缀（参见第 2 章的“ 模块与应用程序 ”一节）。

使用 klogd

klogd 守护进程的近期版本可以在 oops 存放到记录文件前对 oops 消息解码。解码过程只由版本 1.3 或更新版本的守护进程完成，而且只有将-k /usr/src/linux/System.map 做为参数传递给守护进程时才解码。（你可以用其他符号表文件代替 System.map）
有新的 klogd 给出的 faulty 的 oops 如下所示，它写到了系统记录中：
（代码）
我想能解码的 klogd 对于调试一般的 Linux 安装的核心来说是很好的工具。由 klogd 解码的消息包括大部分 ksymoops 的功能，而且也要求用户编译额外的工具，或是，当系统出现故障时，为了给出完整的错误报告而合并两个输出。当 oops 发生在内核时，守护进程还会正确地解码指令指针。它并不反汇编代码，但这不是问题，当错误报告给出消息时，二进制数据仍然存在，可以离线反汇编代码。
守护进程的另一个功能就是，如果符号表版本与当前内核不匹配，它会拒绝解析符号。如果在系统记录中解析出了符号，你可以确信它是正确的解码。
然而，尽管它对 Linux 用户很有帮助，这个工具在调试模块时没有什么帮助。我个人没有在开放软件的电脑里使用解码选项。klogd 的问题是它不解析模块中的符号；因为守护进程在程序员加载模块前就已经运行了，即使读了/proc/ksyms 也不会有什么帮助。记录文件中存在解析后的符号会使 oops 和 ksymoops 混淆，造成进一步解析的困难。
如果你需要使用 klogd 调试你的模块，最新版本的守护进程需要加入一些新的特殊支持，我期待它的完成，只要给内核打一个小补丁就可以了。

系统挂起

尽管内核代码中的大多数错误仅会导致一个 oops 消息，有时它们困难完全将系统挂起。如

果系统挂起了，没有消息能够打印出来。例如，如果代码遇到一个死循环，内核停止了调度过程，系统不会再响应任何动作，包括魔法键 Ctrl-Alt-Del 组合。
处理系统挂起有两个选择——一个是防范与未然，另一个就是亡羊补牢，在发生挂起后调试代码。
通过在策略点上插入 schedule 调用可以防止死循环。schedule 调用（正如你所猜想到的）调用调度器，因此允许其他进程偷取当然进程的 CPU 时间。如果进程因你的驱动程序中的错误而在内核空间循环，你可以在跟踪到这种情况后杀掉这个进程。
在驱动程序代码中插入 schedule 调用会给程序员带来新的“ 问题 ”：函数,，以及调用轨迹中的所有函数，必须是可重入的。在正常环境下，由于不同的进程可能并发地访问设备，驱动程序做为整体是可重入的，但由于 Linux 内核是不可抢占的，不必每个函数都是可重入的。但如果驱动程序函数允许调度器中断当前进程，另一个不同的进程可能会进入同一个函数。如果 schedule 调用仅在调试期间打开，如果你不允许，你可以避免两个并发进程访问驱动程序，所以并发性倒不是什么非常重要的问题。在介绍阻塞型操作时（第 5 章的“ 写可重入代码 ”）我们再详细介绍并发性问题。
如果要调试死循环，你可以利用 Linux 键盘的特殊键。默认情况下，如果和修饰键一起按了 PrScr 键（键码是 70），系统会向当前控制台打印有关机器状态的有用信息。这一功能在 x86 和 Alpha 系统都有。Linux 的 Sparc 移植也有同样的功能，但它使用了标记为“ Break/Scroll Lock ”的键（键码是 30）。
每一个特殊函数都有一个名字，并如下面所示都有一个按键事件与之对应。组合键之后的括号里是函数名。
Shift-PrScr（Show_Memory）
打印若干行关于内存使用的信息，尤其是有关缓冲区高速缓存的使用情况。
Control-PrScr（Show_State）
针对系统里的每一个处理器打印一行信息，同时还打印内部进程树。对当前进程进行标记。
RightAlt-PrScr（Show_Registers）
由于它可以打印按键时的处理器寄存器内容，它是系统挂起时最重要的一个键了。如果有当前内核的系统表的话，查看指令计数器以及它如何随时间变化，对了解代码在何处循环非常有帮助。
如果想将这些函数映射到不同的键上，每一个函数名都可以做为参数传递给 loadkeys。键盘映射表可以任意修改（这是“ 策略无关的 ”）。
如果 console_loglevel 足够到的话，这些函数打印的消息会出现在控制台上。如果不是你运行了一个旧 klogd 和一个新内核的话，默认记录级应该足够了。如果没有出现消息，你可以象以前说的那样提升记录级。“ 足够高 ”的具体值与你使用的内核版本有关。对于 Linux 2.0 或更新的版本来说是 5。
即便当系统挂起时，消息也会打印到控制台上，确认记录级足够高是非常重要的。消息是在产生中断时生成的，因此即便有错的进程不释放 CPU 也可以运行——当然，除非中断被屏蔽了，不过如果发生这种情况既不太可能也非常不幸。
有时系统看起来象是挂起了，但其实不是。例如，如果键盘因某种奇怪的原因被锁住了就会发生这种情况。这种假挂起可以通过查看你为探明此种情况而运行的程序输出来判断。我有一个程序会不断地更新 LED 显示器上的时钟，我发现这个对于验证调度器尚在运行非常有用。你可以不必依赖外部设备就可以检查调度器，你可以实现一个程序让键盘 LED 闪烁，或是不断地打开关闭软盘马达，或是不断触动扬声器——不过我个人认为，通常的蜂鸣声很烦人，应该尽量避免。看看 ioctl 命令 KDMKTONE。O’Reilly FTP 站点上的例子程序（misc-progs/heartbeat.c）中有一个是让键盘 LED 不断闪烁的。

如果键盘不接收输入了，最佳的处理手段是从网络登录在系统中，杀掉任何违例的进程，或是重新设置键盘（用 kdb_mode -a）。然而，如果你没有网络可用来恢复的话，发现系统挂起是由键盘锁死造成的一点儿用也没有。如果情况确实是这样，你应该配置一种替代输入设备，至少可以保证正常地重启系统。对于你的计算机来说，关闭系统或重启比起所谓的按“ 大红钮 ”要更方便一些，至少它可以免去长时间地 fsck 扫描磁盘。
这种替代输入设备可以是游戏杆或是鼠标。在 sunsite.edu.cn 上有一个游戏杆重启守护进程，gpm-1.10 或更新的鼠标服务器可以通过命令行选项支持类似的功能。如果键盘没有锁死，但是却误入“ 原始 ”模式，你可以看看 kdb 包中文档介绍的一些小技巧。我建议最好在问题出现以前就看看这些文档，否则就太晚了。另一种可能是配置 gpm-root 菜单，增添一个“ reboot ”或“ reset keyboard ”菜单项；gpm-root 一个响应控制鼠标事件的守护进程，它用来在屏幕上显示菜单和执行所配置的动作。
最好，你会可以按“ 留意安全键 ”（SAK），一个用于将系统恢复为可用状态的特殊键。由于不是所有的实现都能用，当前 Linux 版本的默认键盘表中没有为此键特设一项。不过你还是可以用 loadkeys 将你的键盘上的一个键映射为 SAK。你应该看看 drivers/char 目录中的 SAK 实现。代码中的注释解释了为什么这个键在 Linux 2.0 中不是总能工作，这里我就不多说了。不过，如果你运行版本 2.1.9 或是更新的版本，你就可以使用非常可靠地留意安全键了。此外，2.1.43 及后续版本内核还有一个编译选项选择是否打开“ SysRq 魔法键 ”；我建议你看一看 drivers/char/sysrq.c 中的代码并使用这项新技术。
如果你的驱动程序真的将系统挂起了，而且你有不知道在哪插入 schedule 调用，最佳的处理方法就是加一些打印消息，并将它们打印到控制台上（通过修改 console_loglevel 变量值）。在重演挂起过程时，最好将所有的磁盘都以只读方式安装在系统上。如果磁盘是只读的或没有安装，就不会存在破坏文件系统或使其进入不一致状态的危险。至少你可以避免在复位系统后运行 fsck。另一中方法就是使用 NFS 根计算机来测试模块。在这种情况下，由于 NFS 服务器管理文件系统的一致性，而它又不会受你的驱动程序的影响，你可以避免任何的文件系统崩溃。

使用调试器

最后一种调试模块的方法就是使用调试器来一步步地跟踪代码，查看变量和机器寄存器的值。这种方法非常耗时，应该尽可能地避免。不过，某些情况下通过调试器对代码进行细粒度的分析是非常有益的。在这里，我们所说的被调试的代码运行在内核空间——除非你远程控制内核，否则不可能一步步跟踪内核，这会使很多事情变得更加困难。由于远程控制很少用到，我们最后介绍这项技术。所幸的是，在当前版本的内核中可以查看和修改变量。
在这一级上熟练地使用调试器需要精通 gdb 命令，对汇编码有一定了解，并且有能够将源码与优化后的汇编码对应起来的能力。
不幸的是，gdb 更适合与调试核心而不是模块，调试模块化的代码需要更多的技术。这更多的技术就是 kdebug 包，它利用 gdb 的“ 远程调试 ”接口控制本地内核。我将在介绍普通调试器后介绍 kdebug。

使用 gdb

gdb 在探究系统内部行为时非常有用。启动调试器时必须假想内核就是一个应用程序。除了指定内核文件名外，你还应该在命令行中提供内存镜象文件的名字。典型的 gdb 调用如下所

示：
（代码）
第一个参数是未经压缩的内核可执行文件（在你编译完内核后，这个文件在/usr/src/linux 目录中）的名字。只有 x86 体系结构有 zImage 文件（有时称为 vmlinuz），它是一种解决 Intel 处理器实模式下只有 640KB 限制的一种技巧；而无论在哪个平台上，vmlinux 都是你所编译的未经压缩的内核。
gdb 命令行的第二个参数是是内存镜象文件的名字。与其他在/proc 下的文件类似，/proc/kcore 也是在被读取时产生的。当 read 系统调用在/proc 文件系统执行时，它映射到一个用于数据生成而不是数据读取的函数上；我们已在“使用/proc 文件系统”一节中介绍了这个功能。系统用 kcore 来表示按内存镜象文件格式存储的内核“可执行文件”；由于它要表示整个内核地址空间，它是一个非常巨大的文件，对应所有的物理内存。利用 gdb，你可以通过标准 gdb 命令查看内核标量。例如，p jiffies 可以打印从系统启动到当前时刻的时钟滴答数。
当你从 gdb 打印数据时，内核还在运行，不同数据项会在不同时刻有不同的数值；然而，gdb 为了优化对内存镜象文件的访问会将已经读到的数据缓存起来。如果你再次查看 jiffies 变量，你会得到和以前相同的值。缓存变量值防止额外的磁盘操作对普通内存镜象文件来说是对的，但对“动态”内存镜象文件来说就不是很方便了。解决方法是在你想刷新 gdb 缓存的时候执行 core-file /proc/kcore 命令；调试器将使用新的内存镜象文件并废弃旧信息。但是，读新数据时你并不总是需要执行 core-file 命令；gdb 以 1KB 的尺度读取内存镜象文件，仅仅缓存它所引用的若干块。
你不能用普通 gdb 做的是修改内核数据；由于调试器需要在访问内存镜象前运行被调试程序，它是不会去修改内存镜象文件的。当调试内核镜象时，执行 run 命令会导致在执行若干指令后导致段违例。出于这个原因，/proc/kcore 都没有实现 write 方法。
如果你用调试选项（-g）编译了内核，结果产生的 vmlinux 比没有用-g 选项的更适合于 gdb。不过要注意，用-g 选项编译内核需要大量的磁盘空间——支持网络和很少几个设备和文件系统的 2.0 内核在 PC 上需要 11KB。不过不管怎样，你都可以生成 zImage 文件并用它来其他系统：在生成可启动镜象时由于选项-g 而加入的调试信息最终都被去掉了。如果我有足够的磁盘空间，我会一致打开-g 选项的。
在非 PC 计算机上则有不同的方法。在 Alpha 上，make boot 会在生成可启动镜象前将调试信息去掉，所以你最终会获得 vmlinux 和 vmlinux.gz 两个文件。gdb 可以使用前者，但你只能用后者启动。在 Sparc 上，默认情况下内核（至少是 2.0 内核）不会被去掉调试信息，所以你需要在将其传递给 silo（Sparc 的内核加载器）前将调试信息去掉，这样才能启动。由于尺寸的问题，无论 milo（Alpha 的内核加载器）还是 silo 都不能启动未去掉调试信息的内核。
当你用-g 选项编译内核并且用 vmlinux 和/proc/kcore 一起使用调试器，gdb 可以返回很多有关内核内部结构的信息。例如，你可以使用类似于这样的命令，p *module_list，p *module_list->next 和 p *chrdevs[4]->fops 等显示这些结构的内容。如果你手头有内核映射表和源码的话，这些探测命令是非常有用的。
另一个 gdb 可以在当前内核上执行的有用任务是，通过 disassemble 命令（它可以缩写）或是“检查指令”（x/i）命令反汇编函数。disassemble 命令的参数可以是函数名或是内存区范围，而 x/i 则使用一个内存地址做为参数，也可以用符号名。例如，你可以用 x/20i 反汇编 20 条指令。注意，你不能反汇编一个模块的函数，这是因为调试器处理 vmlinux，它并不知道你的模块的信息。如果你试图用模块的地址反汇编代码，gdb 很有可能会报告“不能访问 xxxx 处的内存（Cannot access memory at xxxx）”。基于同样的原因，你不查看属于模块的数据项。如果你知道你的变量的地址，你可以从/dev/mem 中读出它的值，但很难弄明白从系

统内存中分解出的数据是什么含义。
如果你需要反汇编模块函数，你最好对用 objdump 工具处理你的模块文件。很不幸，该工具只能对磁盘上的文件进行处理，而不能对运行中的模块进行处理；因此，objdump 中给出的地址都是未经重定位的地址，与模块的运行环境无关。
如你所见，当你的目的是查看内核的运行情况时，gdb 是一个非常有用的工具，但它缺少某些功能，最重要的一些功能就是修改内核项和访问模块的功能。这些空白将由 kdebug 包填补。

使用 kdebug

你可用从一般的 FTP 站点下的 pcmcia/extras 目录下拿到 kdebug，但是如果你想确保拿到的是最新的版本，你最好到 ftp://hyper.stanford.edu/pub/pcmcia/extras/去找。该工具与 pcmcia 没有什么关系，但是这两个包是同一个作者写的。
kdebug 是一个使用 gdb“ 远程调试 ”接口与内核通信的小工具。使用时首先向内核加载一个模块，调试器通过/dev/kdebug 访问内核数据。gdb 将该设备当成一个与被调试“ 应用 ”通信的串口设备，但它仅仅是一个用于访问内核空间的通信通道。由于模块本身运行在内核空间，它可以看到普通调试器无法访问的内核空间地址。正如你所猜想到的，模块是一个字符设备驱动程序，并且使用了主设备号动态分配技术。
kdebug 的优点在于，你无需打补丁或重新编译：无论是内核还是调试器都无需修改。你所需要做的就是编译和安装软件包，然后调用 kgdb，kgdb 是一个完成某些配置并调用 gdb，通过新接口访问内核部件结构的脚本程序。
但是，即便是 kdebug 也没有提供单步跟踪内核代码和设置断点的功能。这几乎是不可避免的，因为内核必须保持运行状态以保证系统的出于运行状态，跟踪内核代码的唯一方法就是后面将要谈到的从另外一台计算机上通过串口控制系统。不过 kgdb 的实现允许用户修改被调试应用（即当前内核）的数据项，可以传递给内核任意数目的参数，并以读写方式访问模块所属的内存区。
最后一个功能就是通过 gdb 命令将模块符号表增加到调试器内部的符号表中。这个工作是由 kgdb 完成的。然后当用户请求访问某个符号时，gdb 就知道它的地址是哪了。最终的访问是由模块里的内核代码完成的。不过要注意，kdebug 的当前版本（1.6）在映射模块化代码地址方面还有些问题。你最好通过打印一些符号并与/proc/ksyms 中的值进行比较来做些检查。如果地址没有匹配，你可以使用数值，但必须将它们强行转换为正确的类型。下面就是一个强制类型转换的例子：
（代码）
kdebug 的另一个强于 gdb 的优点是，它允许你在数据结构被修改后读取到最新的值，而不必刷新调试器的缓存；gdb 命令 set remotecache 0 可以用来关闭数据缓存。
由于 kdebug 与 gdb 使用起来很相似，这里我就不过多地罗列使用这个工具的例子了。对于知道如何使用调试器的人来说，这种例子很简单，但对于那些对调试器一无所知的人来说就很晦涩了。能够熟练地使用调试器需要时间和经验，我不准备在这里承担老师的责任。
总而言之，kdebug是一个非常好的程序。在线修改数据结构对于开发人员来说是一个非常大的进步（而且一种将系统挂起的最简单方法）。现在有许多工具可以使你的开发工作更轻松——例如，在开发scull期间，当模块的使用计数器增长后*，我可以使用kdebug来将其复位为 0。这就不必每次都麻烦我重启机器，登录，再次启动我的应用程序等等。

* 使用计数器在模块地址空间的最开始的部分，不过这是未公开的，以后可能会发生变化。

远程调试

调试内核镜象的最后一个方法是使用 gdb 的远程调试能力。
当执行远程调试的时候，你需要两台计算机：一台运行 gdb；另一台运行你要调试的内核。这两台计算机间用普通串口连接起来。如你所料，控制 gdb 必须能够理解它所控制的内核的二进制格式。如果这两台计算机是不同的体系结构，必须将调试器编译为可以支持目标平台的。
在 2.0 中，Linux 内核的 Intel 版本不支持远程调试，但是 Alpha 和 Sparc 版本都支持。在 Alpha 版本中，你必须在编译时包含对远程调试的支持，并在启动时通过传递给内核命令行参数 kgdb=1 或只有 kgdb 打开这个功能。在 Sparc 上，始终包含了对远程调试的支持。启动选项 kgdb=ttyx 可以用来选择在哪个串口上控制内核，x 可以是 a 或 b。如果没有使用 kgdb=选项，内核就按正常方式启动。
如果在内核中打开了远程调试功能，系统在启动时就会调用一个特殊的初始化函数，配置被调试内核处理它自己的断点，并且跳转到一个编译自程序中的断点。这会暂停内核的正常执行，并将控制转移给断点服务例程。这一处理函数在串口线上等待来自于 gdb 的命令，当它获得 gdb 的命令后，就执行相应的功能。通过这一配置，程序员可以单步跟踪内核代码，设置断点，并且完成 gdb 所允许的其他任务。
在控制端，需要一个目标镜象的副本（我们假设它是 linux.img），还需要一个你要调试的模块副本。如下命令必须传递给 gdb：
file linux.img
file 命令告诉 gdb 哪个二进制文件需要调试。另一种方法是在命令行中传递镜象文件名。这个文件本身必须和运行在另一端的内核一模一样。
target remote /dev/ttyS1
这条命令通知 gdb 使用远程计算机做为调试过程的目标。/dev/ttyS1 是用来通信的本地串口，你可以指定任一设备。例如，前面介绍的 kdebug 软件包中的 kgdb 脚本使用 target remote /dev/kdebug。
add-symbol-file module.o address
如果你要调试已经加载到被控内核的模块的话，在控制系统上你需要一个模块目标文件的副本。add-symbol-file 通知 gdb 处理模块文件，假定模块代码被定位在地址 address 上了。
尽管远程调试可以用于调试模块，但你还是要加载模块，并且在模块上插入断点前还需要触发另一个断点，调试模块还是需要很多技巧的。我个人不会使用远程调试去跟踪模块，除非异步运行的代码，如中断处理函数，出了问题。

第 5 章 字符设备驱动程序的扩展操作

在关于字符设备驱动程序的那一章中，我们构建了一个完整的设备驱动程序，从中用户可以读也可以写。但实际一个驱动程序通常会提供比同步 read 和 write 更多的功能。现在如果出了什么毛病，我已经配备了调试工具，我们可以大胆的实验并实现新操作。
通过补充设备读写操作的功能之一就是控制硬件，最常用的通过设备驱动程序完成控制动作的方法就是实现 ioctl 方法。另一种方法是检查写到设备中的数据流，使用特殊序列做为控制命令。尽管有时也使用后者，但应该尽量避免这样使用。不过稍后我们还是会在本章的“ 非 ioctl 设备控制 ”一节中介绍这项技术。
正如我在前一章中所猜想的，ioctl 系统调用为驱动程序执行“ 命令 ”提供了一个设备相关的入口点。与 read 和其他方法不同，ioctl 是设备相关的，它允许应用程序访问被驱动硬件的特殊功能——配置设备以及进入或退出操作模式。这些“ 控制操作 ”通常无法通过 read/write 文件操作完成。例如，你向串口写的所有数据都通过串口发送出去了，你无法通过写设备改变波特率。这就是 ioctl 所要做的：控制 I/O 通道。
实际设备（与 scull 不同）的另一个重要功能是，读或写的数据需要同其他硬件交互，需要某些同步机制。阻塞型 I/O 和异步触发的概念将满足这些需求，本章将通过一个改写的 scull 设备介绍这些内容。驱动程序利用不同进程间的交互产生异步事件。与最初的 scull 相同，你无需特殊硬件来测试驱动程序是否可以工作。直到第 8 章“ 硬件管理 ”我才会真正去与硬件打交道。

ioctl

在用户空间内调用 ioctl 函数的原型大致如下：
（代码）
由于使用了一连串的“ . ”的缘故，该原型在 Unix 系统调用列表之中非常突出，这些点代表可变数目参数。但是在实际系统中，系统调用实际上不会有可变数目个参数。因为用户程序只能通过第 2 章“ 编写和运行模块 ”的“ 用户空间和内核空间 ”一节中介绍的硬件“ 门 ”才能访问内核，系统调用必须有精确定义的参数个数。因此，ioctl 的第 3 个参数事实上只是一个可选参数，这里用点只是为了在编译时防止编译器进行类型检查。第 3 个参数的具体情况与要完成的控制命令（第 2 个参数）有关。某些命令不需要参数，某些需要一个整数做参数，而某些则需要一个指针做参数。使用指针通常是可以用来向 ioctl 传递任意数目数据；设备可以从用户空间接收任意大小的数据。
系统调用的参数根据方法的声明传递给驱动程序方法：
（代码）
inode 和 filp 指针是根据应用程序传递的文件描述符 fd 计算而得的，与 read 和 write 的用法一致。参数 cmd 不经修改地传递给驱动程序，可选的 arg 参数无论是指针还是整数值，它都以 unsigned long 的形式传递给驱动程序。如果调用程序没有传递第 3 个参数，驱动程序所接收的 arg 没有任何意义。
由于附加参数的类型检查被关闭了，如果非法参数传递给 ioctl，编译器无法向你报警，程序员在运行前是无法注意这个错误的。这是我所见到的 ioctl 语义方面的唯一一个问题。
如你所想，大多数 ioctl 实现都包括一个 switch 语句来根据 cmd 参数选择正确的操作。不同

的命令对应不同的数值，为了简化代码我们通常会使用符号名代替数值。这些符号名都是在预处理中赋值的。不同的驱动程序通常会在它们的头文件中声明这些符号；scull 就在 scull.h 中声明了这些符号。

选择 ioctl 命令

在编写 ioctl 代码之前，你需要选择对应不同命令的命令号。遗憾的是，简单地从 1 开始选择号码是不能奏效的。

为了防止对错误的设备使用正确的命令，命令号应该在系统范围内是唯一的。这种失配并不是不很容易发生，程序可能发现自己正在对象 FIFO 和 kmouse 这类非串口输入流修改波特率。如果每一个 ioctl 命令都是唯一的，应用程序就会获得一个 EINVAL 错误，而不是无意间成功地完成了操作。

为了达到唯一性的目的，每一个命令号都应该由多个位字段组成。Linux 的第一版使用了一个 16 位整数：高 8 位是与设备相关的“幻”数，低 8 位是一个序列号码，在设备内是唯一的。这是因为，用 Linus 的话说，他有点“无头绪”，后来才接收了一个更好的位字段分割方案。遗憾的是，很少有驱动程序使用新的约定，这就挫伤了程序员使用新约定的热情。在我的源码中，为了发掘这种约定都提供了那些功能，同时防止被其他开发人员当成异教徒而禁止，我使用了新的定义命令的方法。

为了给我的驱动程序选择 ioctl 号，你应该首先看看 include/asm/ioctl.h 和 Documentation/ioctl-number.txt 这两个文件。头文件定义了位字段：类型（幻数），基数，传送方向，参数的尺寸等等。ioctl-number.txt 文件中罗列了在内核中使用的幻数。这个文件的新版本（2.0 以及后继内核）也给出了为什么应该使用这个约定的原因。

很不幸，在 1.2.x 中发行的头文件没有给出切分 ioctl 位字段宏的全集。如果你需要象我的 scull 一样使用这种新方法，同时还要保持向后兼容性，你使用 scull/sysdep.h 中的若干代码行，我在那里给出了解决问题的文档的代码。

现在已经不赞成使用的选择 ioctl 号码的旧方法非常简单：选择一个 8 位幻数，比如“k”（十六进制为 0x6b），然后加上一个基数，就象这样：

（代码）

如果应用程序和驱动程序都使用了相同的号码，你只要在驱动程序里实现 switch 语句就可以了。但是，这种在传统 Unix 中有基础的定义 ioctl 号码的方法，不应该再在新约定中使用。这里我介绍就方法只是想给你看看一个 ioctl 号码大致是个什么样子的。

新的定义号码的方法使用了 4 个位字段，它们有如下意义。下面我所介绍的新符号都定义在 <linux/ioctl.h>中。

类型

幻数。选择一个号码，并在整个驱动程序中使用这个号码。这个字段有 8 位宽（_IOC_TYPEBITS）。

号码

基（序列）数。它也是 8 位宽（_IOC_NRBITS）。

方向

如果该命令有数据传输，它定义数据传输的方向。可以使用的值有，_IOC_NONE（没有数据传输），_IOC_READ，_IOC_WRITE 和_IOC_READ | _IOC_WRITE（双向传输数据）。数据传输是从应用程序的角度看的；IOC_READ 意味着从设备中读数据，驱动程序必须向用户空间写数据。注意，该字段是一个位屏蔽码，因此可以用逻辑 AND 操作从中分解出_IOC_READ 和_IOC_WRITE。

尺寸
所涉及的数据大小。这个字段的宽度与体系结构有关，当前的范围从 8 位到 14 位不等。你可以在宏_IOC_SIZEBITS 中找到某种体系结构的具体数值。不过，如果你想要你的驱动程序可移植，你只能认为最大尺寸可达 255 个字节。系统并不强制你使用这个字段。如果你需要更大尺度的数据传输，你可以忽略这个字段。下面我们将介绍如何使用这个字段。
包含在<linux/ioctl.h>之中的头文件<asm/ioctl.h>定义了可以用来构造命令号码的宏：_IO(type,nr)，_IOR(type,nr,size)，_IOW(type,nr,size)和 IOWR(type,nr,size)。每一个宏都对应一种可能的数据传输方向，其他字段通过参数传递。头文件还定义了解码宏：_IOC_DIR(nr)，_IOC_TYPE(nr)，_IOC_NR(nr)和_IOC_SIZE(nr)。我不打算详细介绍这些宏，头文件里的定义已经足够清楚了，本节稍后会给出样例。
这里是 scull 中如果定义 ioctl 命令的。特别地，这些命令设置并获取驱动程序的配置参数。在标准的宏定义中，要传送的数据项的尺寸有数据项自身的实例代表，而不是 sizeof(item)，这是因为 sizeof 是宏扩展后的一部分。
（代码）
最后一条命令，HARDRESET，用来将模块使用计数器复位为 0，这样就可以在计数器发生错误时就可以卸载模块了。实际的源码定义了从 IOCHQSET 到 HARDRESET 间的所有命令，但这里没有列出。
我选择用两种方法实现整数参数传递——通过指针和显式数值，尽管根据已有的约定，ioctl 应该使用指针完成数据交换。同样，这两种方法还用于返回整数：通过指针和设置返回值。如果返回值是正的，这就可以工作；对与任何一个系统调用的返回值，正值是受保护的（如我们在 read 和 write 所见到的），而负值则被认为是一个错误值，用其设置用户空间中的 errno 变量。
“交换”和“移位”操作并不专用于scull设备。我实现“交换”操作是为了给出“方向”字段的所有可能值，而“移位”操作则将“告知”和“查询”操作组合在一起。某些时候是需要原子性*测试兼设置这类操作的——特别是当应用程序需要加锁和解锁时。
显式的命令基数没有什么特殊意义。它只是用来区分命令的。事实上，由于 ioctl 号码的“方向”为会有所不同，你甚至可以在读命令和写命令中使用同一个基数。我选择除了在声明中使用基数外，其他地方都不使用它，这样我就不必为符号值赋值了。这也是为什么显式的号码出现在上面的定义中。我只是向你介绍一种使用命令号码的方法，你可以自由地采用不同的方法使用它。
当前，参数 cmd 的值内核并没有使用，而且以后也不可能使用。因此，如果你想偷懒，你可以省去上面那些复杂的声明，而直接显式地使用一组 16 位数值。但另一方面，如果你这样做了，你就无法从使用位字段中受益了。头文件<linux/kd.h>就是这种旧风格方法的例子，但是它们并不是因为偷懒才这样做的。修改这个文件需要重新编译许多应用程序。

返回值

ioctl 的实现通常就是根据命令号码的一个 switch 语句。但是，当命令号码不能匹配任何一个合法操作时，default 选择使用是什么？这个问题是很有争议性的。大多数内核函数返回-EINVAL（“非法参数”），这是由于命令参数确实不是一个合法的参数，这样做是合适的。然而，POSIX 标准上说，如果调用了一个不合适的 ioctl 命令，应该返回-ENOTTY。对应的

* 当一段程序代码总是被当做一条指令被执行，且不可能在期间发生其他操作，我们就称这段代码是原子性的。

消息是“ 不是终端 ”——这不是用户所期望的。你不得不决定是严格依从标准还是一般常识。我们将本章的后面介绍为什么依从 POSIX 标准需要返回 ENOTTY。

预定义命令

尽管 ioctl 系统调用大部分都用于操作设备，但还有一些命令是由内核识别的。注意，这些命令是在你自己的文件操作前调用的，所以如果你选择了和它们相同的命令号码，你将无法接收到那个命令的请求，而且由于 ioctl 命令的不唯一性，应用程序会请求一些未可知的请求。

预定义命令分为 3 组：用于任何文件（普通，设备，FIFO 和套接字文件）的，仅用于普通文件的以及和文件系统相关的 ;最后一组命令只能在宿主机文件系统上执行(间chattr命令)。设备驱动程序编写者仅对第 1 组感兴趣就可以了，它们的幻数是“ T ”。分析其他组的工作将留做读者的练习；ext2_ioctl 是其中最有意思的函数（尽管比你想象的要容易得多），它实现了只追加标志和不可变标志。

下列 ioctl 命令对任何文件都是预定义的：

FIOCLEX

设置 exec 时关闭标志（File IOctl Close on EXec）。

FIONCLEX

清除 exec 时关闭标志。

FIOASYNC

设置或复位文件的同步写。Linux 中没有实现同步写；但这个调用存在，这样请求同步写的应用程序就可以编译和运行了。如果你不知道同步写是怎么回事，你也不用费神去了解它了：你不需要它。

FIONBIO

“ File IOctl Nonblocking I/O（文件 ioctl 非阻塞型 I/O）”（稍后在“ 阻塞型和非阻塞型操作 ”一节中介绍）。该调用修改 filp->f_flags 中的 O_NONBLOCK 标志。传递给系统调用的第 3 个参数用来表明该标志是设置还是清除。我们将在本章后面谈到它的作用。注意，fcntl 系统调用使用 F_SETFL 命令也可以修改这个标志。

列表中的最后一项引入了一个新系统调用 fcntl，它看起来和 ioctl 很象。事实上，fcntl 调用与 ioctl 非常相似，它也有一个命令参数和额外（可选的）一个参数。它和 ioctl 分开主要是由于历史原因：当 Unix 开发人员面对“ 控制 ”I/O 操作的问题时，他们决定文件和设备应该是不同的。那时，唯一的设备是终端，这也就解释了为什么-ENOTTY 是标准的非法 ioctl 命令的返回值。这个问题是是否保持向后兼容性的老问题。

使用 ioctl 参数

我们需要讲解的最后一点是，在分析 scull 驱动程序的 ioctl 代码前，首先弄明白如何使用那个额外的参数。如果它是一个整数就非常简单了：可以直接使用它。如果它是一个指针，就必须注意一些问题了。

当用一个指针引用用户空间时，我们首先要确保它指向了合法的用户空间，并且对应页面当前恰在映射中。如果内核代码试图访问范围之外的地址，处理器就会发出一个异常。内核代码中的异常将由上至 2.0.x 的内核转换为 oops 消息。设备驱动程序应该通过验证将要访问的用户地址空间的合法性来防止这种失效的发生，如果地址是非法的应该返回一个错误码。

Linux 2.1 中引入新功能之一就是内核代码的异常处理。遗憾的是，正确的实现需要驱动程序-内核接口的较大改动。本章给出的非法只适用于旧内核，从 1.2.13 到 2.0.x。新接口将在第 17 章“ 近期发展 ”的“ 处理内核空间失效 ”一节中介绍，那里给出的例子通过某些预处理宏将使支持的内核扩展到 2.1.43。

内核 1.x.y 和 2.0.x 的地址验证是通过函数 verify_area 实现的，它的原型定义在<linux/mm.h>中：

（代码）

第一个参数应该是 VERIFY_READ 或 VERIFY_WRITE，这取决于你要在内存区上完成读还是写操作。ptr 参数是一个用户空间地址，extent 是一个字节计数。例如，如果 ioctl 需要从用户空间读一个整数，extent 就是 sizeof(int)。如果在指定的地址上进行读和写操作，使用 VERIFY_WRITE，它是 VERIFY_READ 的超集。

验证读只检查地址是否是合法的：除此之外，验证写要好检查只读和 copy-on-write 页面。copy-on-write 页面一个共享可写页面，它还没有被任何共享进程写过；当你验证写时，verify_area 完成“ 复制兼完成可写配置 ”操作。很有意思的是，这里无需检查页面是否“ 在 ”内存中，这是由于合法页面将由失效函数正确地进行处理，甚至从内核代码中调用也可以。我们已经在第 3 章“ 字符设备 ”的“ Scull 的内存使用 ”一节中看到内核代码可以成功地完成页面失效处理。

象大多数函数一样，verify_area 返回一个整数值：0 意味着成功，负值代表一个错误，应该将这个错误返回给调用者。

scull 源码在 switch 之前分析 ioctl 号码的各个位字段：

（代码）

在调用 verify_area 之后，再有驱动程序完成真正的数据传送。除了 memcpy_tofs 和 memcpy_fromfs 函数外，程序员还可以使用两个专为常用数据尺寸（1，2 和 4 个字节，在以及 64 位平台上的 8 个字节）优化的函数。这些函数定义在<asm/segment.h>中。

put_user(datum, ptr)

实际上它是一个最终调用__put_user 的宏；编译时将其扩展为一条机器指令。驱动程序应该尽可能使用 put_user，而不是 memcpy_tofs。由于在宏表达式中不进行类型检查，你可以传递给 put_user 任何类型的数据指针，不过它应该是一个用户空间地址。数据传输的尺寸依赖于 ptr 参数的类型，这是在编译时通过特殊的 gcc 伪函数实现的，这里没有介绍的必要。结果，如果 ptr 是一个字符指针，就传递 1 个字节，依此类推分别有 2，4 和 8 个字节。如果被指引的数据不是所支持的尺寸，被编译的代码就会调用函数 bad_user_access_length。如果这些编译代码是一个模块，由于这个符号没有开放，模块就不能加载了。

get_user(ptr)

这个宏用来从用户空间获取一个数据。除了数据传输的方向不同外，它与 put_user 是一样的。当 insmod 不能解析符号时，bad_user_access_length 的又臭又长的名字可以当作一个很有意义的错误信息。这样，开发人员就可以在向大众分布模块前加载和测试模块，他会很快找到并修改错误。相反，如果使用了不正确尺寸的 put_user 和 get_user 直接编译到了内核中，bad_user_access_length 就会导致系统 panic。尽管对于尺寸错误的数据传输来说，oops 比其系统 panic 要友好得多，但还是选择了较为激进的方法来尽力杜绝这种错误。

scull 的 ioctl 实现只传送设备的可配置参数，其代码非常简单，罗列如下：

（代码）

还有 6 项是操作 scull_qset 的。这些操作 scull_quantum 的一样，为了节省空间，没有在上面的例子中列出。

从调用者的角度看（即从用户空间），传递和接收参数的 6 种方法如下所示：

（代码）
如果你需要写一个可以在Linux 1.2里运行的模块，get_user和put_user会是非常棘手的函数，因为它们直到内核 1.3 才引入到系统中。在切换到类型依赖宏之前，程序员使用一些称为get_user_byte 等等的函数。旧的宏只在内核 1.3 中定义了，在 2.0 内核中，只有你事先使用了#define WE_REALLY_WANT_TO_USE_A_BROKEN_INTERFACE 时才能使用旧的宏。不过为了可移植性，为旧内核定义 put_user 是一种更好的解决方法，于是为了驱动程序可以在旧内核中良好运行，scull/sydep.h 包含了这些宏的定义。

非 ioctl 设备控制

有时通过向设备自身发送写序列能够更好地完成对设备的控制。例如，这一技术使用在控制台驱动程序中，它称为“ escape 序列 ”，用来控制光标移动，改变默认颜色，或是完成某些配置任务。用这种方法实现设备控制的好处是，用户仅用写数据就可以完成对设备的控制，无需使用（有时是写）完成设备配置的程序。
例如，程序 setterm 通过打印 escape 序列完成对控制台（或其他终端）的配置。这种方法的优点是可以远程控制设备。由于可以简单地重定向数据流完成配置工作，控制程序可以运行在另外一台不同的计算机上，而不一定非要在被控设备的计算机上。你已经在终端上使用了这项技术，但这项技术可以更通用一些。
“ 通过打印控制 ”的缺点是，它给设备增加了策略限制；例如，只有你确认控制序列不会出现在正常写到设备的数据中时，这项技术才是可用的。对于终端来说，这只能说是部分正确。尽管文本显示只意味着显示ASCII字符，但有时控制字符也会出现在正在打印的数据中，因此会影响控制台的配置。例如，当你对二进制文件进行grep时可能会发生这样的情况；分解出的字符行可能什么都包含，最后经常会造成控制台的字体错误*。
写控制特别适合这样的设备，不传输数据，仅相应命令，如机器人设备。
例如，我所写的驱动程序之一是控制一个在两个轴上的摄像头的移动。在这个驱动程序中，“ 设备 ”是一对旧的步进马达，它既不能读也不能写。向步进马达“ 发送数据流 ”多少没有多大意义。在这种情况下，驱动程序将所写的数据解释为 ASCII 命令，并将请求转换为脉冲，实现对步进马达的控制。命令可以是任何象“ 向左移动 14 步 ”,“ 达到位置 100，43 ”或“ 降低默认速度 ”之类的字串。驱动程序仅将/dev 中的设备节点当作为应用程序设立的命令通道。对该设备直接控制的优点是，你可以使用 cat 来移动摄像头，而无需写并编译发出 ioctl 调用的特殊代码。
当编写“ 面向命令的 ”驱动程序时，没有理由要实现 ioctl 方法。为解释器多实现一条命令对于实现和使用来说，都更容易。
好奇的读者可以看看由 O’Reilly FTP 站点提供的源码 stepper 目录中的 stepper 驱动程序；由于我认为代码没有太大的意义（而且质量也不是太高），这里没有包含它。

阻塞型 I/O

read 的一个问题是，当尚未有数据可读，而又没有到文件尾时如何处理。
默认的回答是，“ 我们必须睡眠等待数据。”本节将介绍进程如何睡眠，如何唤醒，以及一个

* Ctrl-N设置替代字体，它有图形字符组成，因此你的外壳的输入来说是不友好的；如果你遇见了这样的问题，回应一个Ctrl-O字符来恢复主字体。

应用程序如何在不阻塞 read 调用的情况下，查看是否有数据。对于写来说也可以适用同样的方法。
通常，在我向你介绍真实的代码前，我将解释若干概念。

睡眠和唤醒

当进程等待事件（可以是输入数据，子进程的终止或是其他什么）时，它需要进入睡眠状态以便其他进程可以使用计算资源。你可以调用如下函数之一让进程进入睡眠状态：
（代码）
然后用如下函数之一唤醒进程：
（代码）
在前面的函数中，wait_queue 指针的指针用来代表事件；我们将在“ 等待队列 ”一节中详细讨论这个数据结构。从现在开始，唤醒进程需要使用进程睡眠时使用的同一个队列。因此，你需要为每一个可能阻塞进程的事件对应一个等待队列。如果你管理 4 个设备，你需要为阻塞读预备 4 个等待队列，为阻塞写再预备 4 个。存放这些队列的最佳位置是与每个设备相关的硬件数据结构（在我们的例子中就是 Scull_Dev）。
但“ 可中断 ”调用和普通调用有什么区别呢？
sleep_on 不能信号取消，但 interruptible_sleep_on 可以。其实，仅在内核的临界区才调用 sleep_on；例如，当等待从磁盘上读取交换页面时。没有这些页面进程就无法继续运行，用信号打断这个操作是没有任何意义的。然而，在所谓“ 长系统调用 ”，如 read，中要使用 interruptible_sleep_on。当进程正等待键盘输入时，用一个信号将进程杀死是很有意义的。
类似地，wake_up 唤醒睡在队列上的任何一个进程，而 wake_up_interruptible 仅唤醒可中断进程。
做为一个驱动程序编写人员，由于进程仅在 read 或 write 期间才睡眠在驱动程序代码上，你应该调用 interruptible_sleep_on 和 wake_up_interruptible。不过，事实上由于没有“ 不可中断 ”的进程在你的队列上睡眠，你也可以调用 wake_up。但是，出于源代码一致性的考虑，最好不这样做。（此外，wake_up 比它的搭档来说要稍微慢一点。）

编写可重入的代码

当进程睡眠后，驱动程序仍然活着，而且可以由另一个进程调用。让我们一控制台驱动程序为例。当一个应用在 tty1 上等待键盘输入，用户切换到 tty2 上并派生了一个新的外壳。现在，两个外壳都在控制台驱动程序中等待键盘输入，但它们睡在不同的队列上：一个睡在与 tty1 相关的队列上，一个睡在与 tty2 相关的队列上。每个进程都阻塞在 interruptible_sleep_on 函数中，但驱动程序让可以继续接收和响应其他 tty 的请求。
可以通过编写“ 可重入代码 ”轻松地处理这种情况。可重入代码是不在全局变量中保留状态信息的代码，因此能够管理交织在一起的调用，而不会将它们混淆起来。如果所有的状态信息都与进程有关，就不会发生相互干扰。
如果需要状态信息，既可以在驱动程序函数的局部变量中保存（每个进程都有不同的堆栈来保存局部变量），也可以保存在访问文件用的filp中的private_data中。由于同一个filp可能在两个进程间共享（通常是父子进程），最好使用局部变量*。

* 注意，内核栈无法存储太大的数据项。在这种情况下，我建议为大数据分配内存，并将这些空间的地址

如果你需要保存大规模的状态信息，你可以将指针保存在局部变量中，并用 kmalloc 获取实际存储空间。此时，你千万别忘了 kfree 这些数据，因为当你在内核空间工作时，没有“在进程终止时释放所有资源”的说法。
你需要将所有调用了 sleep_on（或是 schedule）的函数写成可重入的，并且包括所有在这个函数调用轨迹中的所有函数。如果 sample_read 调用了 sample_getdata，后者可能会阻塞，由于调用它们的进程睡眠后无法阻止另一个进程调用这些函数，sample_read 和 sample_gendata 都必须是可重入的。此外，任何在用户空间和内核空间复制数据的函数也必须是可重入的，这是因为访问用户空间可能会产生页面失效，当内核处理失效页面时，进程可以会进入睡眠状态。

等待队列

我听见你在问的下一个问题是，“我到底如何使用等待队列呢？”
等待队列很容易使用，尽管它的设计很是微妙，但你不需要直到它的内部细节。处理等待队列的最佳方式就是依照如下操作：

- 声明一个 struct wait_queue *变量。你需要为每一个可以让进程睡眠的事件预备这样一个变量。这就是我建议你放在描述硬件特性数据结构中的数据项。
- 将该变量的指针做为参数传递给不同的 sleep_on 和 wake_up 函数。

这相当容易。例如，让我们想象一下，当进程读你的设备时，你要让这个进程睡眠，然后在某人向设备写数据后唤醒这个进程。下面的代码就可以完成这些工作：
（代码）
该设备的这段代码就是例子程序中的 sleepy，象往常一样，可以用 cat 或输入/输出重定向等方法测试它。
上面列出的两个操作是你唯一操作在等待队列上的两个操作。不过，我知道某些读者对它的内部结构感兴趣，但通过源码掌握它的内部结构很困难。如果你不对更多的细节感兴趣，你可以跳过下一节，你不会损失什么的。注意，我谈论的是“当前”实现（版本 2.0.x），但没有什么规定限制内核开发人员必须依照那样的实现。如果出现了更好的实现，内核很容易就会使用新的，由于驱动程序编写人员只能通过那两个合法操作使用等待队列，对他们来说没有什么坏的影响。
当前 struct wait_queue 额实现使用了两个字段：一个指向 struct task_struct 结构（等待进程）的指针，和一个指向 struct wait_queue（链表中的下一个结构）的指针。等待队列是循环链表，最后一个结构指向第一个结构。
该设计的引入注目的特点是，驱动程序编写人员从来不声明或使用这个结构；他们仅仅传递它的指针或指针的指针。实际的结构是存在的，但只在一个地方：在__sleep_on 函数的局部变量中，上面介绍的两个 sleep_on 函数最终会调用这个函数。
这看上去有点奇怪，不过这是一个非常明智的选择，因为无需处理这种结构的分配和释放。进程每次睡在某个队列上，描述其睡眠的数据结构驻留在进程对应的不对换的堆栈页中。
当进程加入或从队列中删除时，实际的操作如图 5-1 所示。
（图 5-1 等待队列的工作示意）

保存在局部变量中。

阻塞型和非阻塞型操作

在分析功能完整的 read 和 write 方法前，我们还需要看看另外一个问题，这就是 filp->f_flags 中的 O_NONBLOCK 标志。这个标志定义在<linux/fcntl.h>中，在最近的内核中，这个头文件由<linux/fs.h>自动包含了。如果你在内核 1.2 中编译你的模块，你需要手动包含 fcntl.h。这个标志的名字取自“ 打开-非阻塞 ”，因为这个标志可以在打开时指定（而且，最初只能在打开时指定）。由于进程在等待数据时的正常行为就是睡眠，这个标志默认情况下是复位的。在阻塞型操作的情况下，应该实现下列操作：

- 如果进程调用 read，但（尚）没有数据，进程必须阻塞。当数据到达时，进程被唤醒，并将数据返回给调用者，即便少于方法的 count 参数中所请求的数据量，也是如此。
- 如果进程调用了 write，缓冲区又没有空间，进程也必须阻塞，而且它必须使用与用来实现读的等待队列不同的等待队列。当数据写进设备后，输出缓冲区中空出部分空间，唤醒进程，write 调用成功完成，如果缓冲区中没有请求中 count 个字节，则进程可能只是完成了部分写。

前面的列表的两个语句都假设，有一个输入和输出缓冲区，而且每个设备驱动程序都有一个。输入缓冲区需要用来在数据达到而又没有人读时避免丢失数据，输出缓冲区用来尽可能增强计算机的性能，尽管这样做不是严格必须的。由于如果系统调用不接收数据的话，数据仍然保存在用户空间的缓冲区中，write 中可以丢失数据。

在驱动程序中实现输出缓冲区可以获得一定的性能收益，这主要是通过较少了用户级/内核级转换和上下文切换的数目达到的。如果没有输出缓冲区（假设是一个慢设备），每次系统调用只接收一个或很少几个字节，并且当进程在 write 中睡眠时，另一进程就会运行（有一次上下文切换）。当第一个进程被唤醒后，它恢复运行（又一次上下文切换），write 返回（内核/用户转换），进程还要继续调用系统调用写更多的数据（内核/用户转换）；然后调用再次被阻塞，再次进行整个循环。如果输出缓冲区足够大，write 首次操作时就成功了；数据在中断时被推送给设备，而不必将控制返回用户空间。适合于设备的输出缓冲区的尺寸显然是和设备相关的。

我们没有在 scull 中使用输入缓冲区，这是因为当调用 read 时，数据已经就绪了。类似地，也没有使用输出缓冲区，数据简单地复制到设备对应的内存区中。我们将在第 9 章“ 中断处理 ”的“ 中断驱动的 I/O ”一节中介绍缓冲区的使用。

如果设置了 O_NONBLOCK 标志，read 和 write 的行为是不同的。此时，如果进程在没有数据就绪时调用了 read，或者在缓冲区没有空间时调用了 write，系统简单地返回-EAGAIN。如你所料，非阻塞型操作立即返回，允许应用查询数据。当使用 stdio 函数处理非阻塞型文件时，由于你很容易误将非阻塞返回认做是 EOF，应用程序应该非常小心。你必须始终检查 errno。

你也许可以从它的名字猜到，O_NONBLOCK 在 open 方法也可有作用。当 open 调用可能会阻塞很长时间时，就需要 O_NONBLOCK 了；例如，当打开一个 FIFO 文件而又（尚）无写者时，或是访问一个被锁住的磁盘文件时。通常，打开设备成功或失败，无需等待外部事件。但是，有时打开设备需要需要很长时间的初始化，你可以选择打开 O_NONBLOCK，如果设置了标志，在设备开始初始化后，会立即返回一个-EAGAIN（再试一次）。你还可以为支持访问策略选择实现阻塞型 open，方式与文件锁类似。我们稍后将在“ 替代 EBUSY 的阻塞型打开 ”一节中看到这样一种实现。

只有 read，write 和 open 文件操作受非阻塞标志的影响。

样例实现：scullpipe

/dev/scullpipe 设备（默认有 4 个设备）是 scull 模块的一部分，用来展示如何实现阻塞型 I/O。在驱动程序内部，阻塞在 read 调用的进程在数据达到时被唤醒；通常会发出一个中断来通知这样一种事件，驱动程序在处理中断时唤醒进程。由于你应该无需任何特殊硬件——没有任何中断处理函数，就可以在任何计算机上运行 scull，scull 的目标与传统驱动程序完全不同。我选择的方法是，利用另一个进程产生数据，唤醒读进程；类似地，用读进程唤醒写者。这种实现非常类似与一个 FIFO（或“命名管道”）文件系统节点的实现，设备名就出自此。设备驱动程序使用一个包含两个等待队列和一个缓冲区的设备结构。缓冲区的大小在通常情况下是可以配置的（编译时，加载时和运行时）。
（代码）
read 实现管理阻塞型和非阻塞型数据，如下所示：
（代码）
如你所见，我在代码中留下了 PDEBUG 语句。当你编译驱动程序时，你可以打开消息，这样就可以更容易地看到不同进程间的交互了。
跟在 interruptible_sleep_on 后的 if 语句处理信号处理。这条语句保证对信号恰当和预定的处理过程，它会让内核完成系统调用重启或返回-EINTR（内核在内部处理-ERESTARTSYS，最终返回到用户空间的是-EINTR）。我不想让内核对阻塞信号完成这样的处理，主要时我想忽略这些信号。否则，我们可以返回-ERESTARTSYS 错误给内核，让它完成它的处理工作。我们将在所有的 read 和 write 实现中使用一样的语句进行信号处理。
write 的实现与 read 非常相似。它唯一的“特殊”功能时，它从不完全填充缓冲区，总时留下至少一个字节的空洞。因此，当缓冲区空的时候，wp 和 rp 时相等的；当存在数据时，它们是不等的。
（代码）
正如我所构想的，设备没有实现阻塞型 open，这要比实际的 FIFO 要简单得多。如果你想要看看实际的代码，你可以在内核源码的 fs/pipe.c 中找到那些代码。
要测试 scullpipe 设备的阻塞型操作，你可以在其上运行一些应用，象往常一样，可以使用输入/输出重定义等方法。由于普通程序不执行非阻塞型操作，测试非阻塞活动要麻烦些。misc-progs 源码目录中包含了一个很简单的程序，称为 nbtest，用它来测试非阻塞型操作，该程序罗列如下。它所做的就是使用非阻塞型 I/O 复制它的输入和输出，并在期间稍做延迟。延迟时间可以通过命令行传递，默认情况下时 1 秒钟。
（代码）

Select

在使用非阻塞型 I/O 时，应用程序经常要利用 select 系统调用，当涉及设备文件时，它依赖于一个设备方法。这个系统调用还用来实现不同源输入的多路复用。在下面的讨论中，我假设你知道在用户空间中 select 的语义的用法。注意，内核 2.1.23 引入了 poll 系统调用，因此为了支持这两个系统调用，它改变驱动程序方法的工作方式。
为了保存所有正在等待文件（或设备）的信息，Linux 2.0 的 select 系统调用的实现使用了 select_table 结构。再次提醒你，你无需了解它的内部结构（但不管怎样，我们一会会稍做介绍），而且只允许调用操作该结构的函数。
当 select 方法发现无需阻塞时，它返回 1；当进程应该等待，它应该“几乎”进入睡眠状态。

在这种情况下，要在 select_table 结构中加入等待队列，并且返回 0。
仅当选择的文件中没有一个可以接收或返回数据时，进程才真正进入睡眠状态。这一过程发生在 fs/select.c 的 sys_select 中。
写 select 操作的代码要比介绍它要容易得多，现在就可以 scull 中时如何实现的：
（代码）
这里没有代码处理‘ 第 3 种形式的选择 ”,选择异常。这种形式的选择时通过 mode == SEL_EX 标别的，但大多数时候你都将其编写为默认情况，在其他选择均失败时执行。异常事件的含义与设备有关，所以你可以选择是否在你自己的驱动程序中实现它们。这种功能将只会为专为你的驱动程序设计的程序使用，但那并不它的初衷。在这方面，它与依赖于设备的 ioctl 调用很相似。在实际使用中，select 中异常条件的主要用途是，通知网络连接上带外（加急）数据的达到，但它还用在终端层以及管道/FIFO 实现中（你可以查看 fs/pipe.c 中的 SEL_EX）。不过要注意，其他 Unix 系统的管道和 FIFO 没有实现异常条件。
这里给出的 select 代码缺少对文件尾的支持。当 read 调用达到文件尾时，它应该返回 0，select 必须通过通告设备可读来支持这种行为，这样应用程序就不会永远等待调用 read 了。例如，在实际的 FIFO 中，当所有的写者都关闭了文件时，读者会看到文件尾，而在 scullpipe 中，读者永远也看不到文件尾。设计这种不同行为的原因时，一般将 FIFO 当做两个进程间的通信通道，而 scullpipe 是一个只要至少有一个读者，所有人就都可以输入数据的垃圾筒。此外，也没有必要重新实现内核中已经有了的设备。
象 FIFO 那样实现文件尾意味着要在 read 和读 select 中检查 dev->nwriters，并做相应处理。不过很遗憾，如果读者在写者前打开设备，它马上就看到文件尾了，没有机会等待数据到达。最好的解决这个问题的方法是，实现阻塞型 open，这个任务做为练习留给读者。

与 read 和 write 的交互

select 调用的目的是事先判断是否有 I/O 操作会阻塞。从这个方面说，它时对 read 和 write 的补充。由于 select 可以让驱动程序同时等待多个数据流（但这与这里的情况无关），select 在这方面也时很有用途的。
为了让应用正确工作，正确实现这 3 个调用时非常重要的。尽管下面的规则已经多多少少谈过了一些，我还要在这里再总结一下。

从设备读取数据

如果在输入缓冲区中有数据，即便比所请求的数据少，而且驱动程序可以保证剩下的数据会马上达到，read 调用应该不经过任何可以察觉的延迟立即返回。如果你至少可以返回 1 个字节，而且很方便的话，你总可以返回比请求少的数据（我们在 scull 就是这样做的）。当前内核中总线鼠标的实现在这方面就时错的，某些程序（如 dd）无法正确读取这些设备。
如果输入缓冲区中没有数据，在至少有一个字节可读前 read 必须阻塞，除非设置了 O_NONBLOCK。非阻塞型 read 立即返回-EAGAIN（尽管在这种情况下某些旧的 System V 会返回 0）。在至少有一个字节可读前，select 必须报告设备不可读。只要有数据可读，我们就使用上一条规则。
如果我们到了文件尾，，无论是否有 O_NONBLOCK，read 都应该立即返回 0。select 应该报告说文件可读。

向设备写数据

如果输出缓冲区有空间，write 应该不做任何延迟返回。它可以接收少于请求数目的数据，但是它须接收至少一个字节。在这种情况下，select 应该报告设备可写。
如果输出缓冲区是满的，在空间释放前 write 一直阻塞，除非设置了 O_NONBLOCK 标志。非阻塞型 write 立即返回，返回值为-EAGAIN（或者在某些条件为 0，如前面旧版本的 System V 所说）。select 应该报告文件不可写。但另一方面，如果设备不能接收任何数据，无论是否设置了 O_NONBLOCK，write 都返回-ENOSPC（“ 设备无可用空间 ”）。
如果使用设备的程序需要确保等候在输出队列中的数据真的完成了传送，驱动程序必须提供一个 fsync 方法。例如，可移动设备使用提供 fsync 入口点。千万不要在调用返回前让 write 调用等待数据传送结束。这是因为，需要应用程序都可用 select 检查设备是否时可以写的。如果设备报告可以写，write 调用应该保持一致，不能阻塞。

刷新待处理输出

我们已经看到 write 方法为什么不能满足所有数据输出的需求。通过同名系统调用调用的 fsync 函数弥补了这一空缺。
如果某些应用需要确保数据传送到设备上，设备就必须实现 fsync 方法。无论是否设置了 O_NONBLOCK 标志，fsync 调用应该仅在设备已经完全刷新数据后才能返回，甚至花些时间也要如此。
fsync 方法没有什么不寻常的功能。调用不是时间关键的，所以每个设备驱动程序都可以按照作者的风格实现这个方法。大多数时候，字符设备驱动程序的 fsync 在其 fops 结构都对应一个 NULL 指针。而块设备总是通过调用通用的 block_fsync 函数实现这个方法，block_fsync 刷新设备的所有缓冲块，一直等到 I/O 结束。

所使用的数据结构

内核 2.0 所使用的 select 特殊实现相当高效，而且还有点复杂。如果你不对操作系统的细节感兴趣，你可以直接跳到下一节。
首先，我建议你看一看图 5-2，它展示了调用 select 所涉及的步骤。看看这张图会有助于搞清除下面的讨论。
select 的工作是由函数 select_wait 和 free_wait 完成的，select_wait 是声明在<linux/sched.h>里的内嵌函数，而 free_wait 则是在 fs/select.c 中定义的。它们使用的数据结构时 struct select_table_entry 数组，每一项都是由 struct wait_queue 和 struct wait_queue **组成的。前者是插入到设备等待队列的实际数据结构（当调用 sleep_on 时以局部变量形式存在的数据结构），而后者是在所选条件有一个为真将当前进程从队列中删除时所需要的“ 句柄 ”——例如，当选择 scullpipe 进行读操作时，它包含&dev->inq（见“ Select ”一节中较早的例子）。简而言之，select_wait 将下一个空闲的 select_table_entry 插入到指定的等待队列中。当系统调用返回时，free_wait 利用对应的指针删除自己等待队列中的每一项。
select_table 结构（有一个指向数据的指针和活动数据变量组成）在 do_select 声明为一个局部变量，这一点与__sleep_on 相似。但数组项却保存在另一个页面上，否则它就会造成当前进程的堆栈溢出。

如果你对这些描述理解起来较为困难，看看源码。一旦你理解了实现，你就可以看到它时如此的简洁和高效。

异步触发

尽管大多数时候阻塞型和非阻塞型操作的组合，以及 select 方法可以有效地查询设备，但某些时候用这种技术管理就不够高效了。例如，想我们想象一下，一个在低优先级执行长计算循环的进程，但它需要尽可能快地处理输入的数据。如果输入通道时键盘，你可以想进程发送信号（使用“ INTR ”字符，一般就是 Ctrl-C），但是这种信号是 tty 层的一部分，在一般字符设备设备中没有。我们所需要的异步触发与此有些不同。此外，任何输入数据都应该产生一个中断，而不仅仅是 Ctrl-C。

为了打开文件的异步触发机制，用户程序必须执行两个步骤。首先，它们指定进程是文件的“ 属主 ”。文件属主的用户 ID 保存在 filp->f_owner 中，可以通过 fcntl 系统调用的 F_SETOWN 命令设置这个值。此外，为了确实打开异步触发机制，用户程序还必须通过另一个 fcntl 命令设置设备的 FASYNC 标志。

在完成这两个步骤后，无论何时新数据到达，输入文件都产生一个 SIGIO 信号。信号发送给存放在 filp->f_owner 的进程（如果是负值，则是进程组）。

例如，如下这些行代码打开 stdin 输入文件到当前进程的异步触发机制：

（代码）

源码中的名为 asynctest 的程序就是这样读取 stdin 的程序。它可以用来测试 scullpipe 的异步功能。这个程序类似与 cat，但在文件尾时不会终止；它只对输入反应，而不是在没有输入时反应。

不过要注意，并不是所有的设备都支持异步触发，而且你可以选择支持或不支持。应用通常应该假设仅有套接字和终端才有异步能力。例如，至少对当前内核来说，管道和 FIFO 就不支持异步触发。鼠标提供了异步触发（尽管在 1.2 中没有），这是因为某些程序希望鼠标能够象 tty 那样发送 SIGIO。

对于输入触发来说还存在一个问题。当进程接收到 SIGIO 后，它不知道时哪个文件有新输入了。如果多于一个文件可以异步触发进程，通知有数据待处理，应用程序仍须借助 select 探测到底发生了什么。

从驱动程序的角度看

与我们更相关的一个话题是，设备驱动程序如何实现异步信号。下面的列表从内核角度给出了这种操作的详细过程：

- 当调用 F_SETOWN 时，除了对 filp->f_owner 赋值以外什么也不做。
- 当调用 F_SETFL 打开 FASYNC 标志时，驱动程序的 fasync 方法被调用。无论 FASYNC 的值何时发生变化，该方法都被调用，通知驱动程序该标志的变化，以便驱动程序能够正确地响应。在文件被打开时，这个标志默认时被清 0 的。我们一会就可以看到这个驱动程序方法的标准实现。
- 当数据达到时，想所有注册异步触发的进程发送 SIGIO 信号。

尽管实现的第一步很简单——在驱动程序端没有什么可做的——其他步骤则为了跟踪不同的异步读者，要涉及一个动态数据结构；同时可能有多个读者。然而，这个动态数据结构不依赖与某个特定设备，内核提供了一套合适的通用实现方法，你不必在每个驱动程序都重写

一遍代码了。
遗憾的是，内核 1.2 没有包含这个实现。对与旧版本的内核来说，在模块中实现异步触发不是很容易，你得写你自己的数据结构。为了简单，scull 模块不对旧内核提供异步触发机制。由 Linux 提供的通用实现基于一个数据结构和两个函数（要在上面谈的步骤中调用）。声明相关内容的头文件是<linux/fs.h>——没什么新的——数据结构称为 struct fasync_struct。如我们处理等待队列一样，我们需要在设备相关的数据结构中插入这个结构的指针。事实上，我们已经在“ 样例实现：scullpipe ”一节中看到了这个字段。
根据如下原型调用那两个函数：
（代码）
当打开文件的 FASYNC 标志被修改时，调用前者从感兴趣进程列表上增加或删除文件，当数据达到时，则应该调用后者。
这里时 scullpipe 如何实现 fasync 方法的：
（代码）
很清除，所有的工作都是由 fasync_helper 完成的。然后，不可能不在驱动程序的方法中实现这一功能，这是因为助手函数需要访问指向 struct fasync_struct *的正确指针（这里是&dev->fasync_queue），而且只有驱动程序才能提供这些信息。
当数据到达时，必须执行下面的语句异步通知读者。由于 scullpipe 读者的新数据是由另一个调用 write 的进程产生的，这条语句出现 scullpipe 的 write 方法中。
（代码）
似乎我们已经完成了所有的工作，但还忘了一件事。我们必须在文件关闭时调用我们的 fasync 方法，去除活动异步读者链表中去除被关闭的文件。尽管这个调用仅当 filp->f_flags 设置为 FASYNC 才需要，调用这个函数不会有什么危害，而且它时比较普遍的实现。例如，如下代码行就是 scullpipe 的 close 方法的一部分：
（代码）
异步触发机制使用的数据结构与 struct wait_queue 结构非常相似，因为两者都涉及等待事件。不同之处是，前者使用 struct file 代替了 struct task_struct。为了向进程发送信号，通过队列中的 struct file 获取 f_owner 字段。

定位设备

本章的难点都以讲过了，下面我们将浏览一下 lseek 方法，它很有用而且很容易实现。注意，在 2.1.0 中该方法的原型稍有变化，第 17 章的“ 原型区别 ”一节中将详细介绍。

lseek 实现

我已经说了，如果设备操作中没有 lseek，内核响应的默认实现是通过修改 filp->f_pos，从文件的起始处和当前位置开始定位。
如果对你的设备来说，相对文件尾定位有意义，你应该提供自己的方法，它看上去就象如下这些代码：
（代码）
这里，唯一的设备相关操作是从设备中获取文件长度。不过为了能够让 lseek 系统调用正确工作，无论何时发生了数据传送，read 和 write 调用都必须相应更新 filp->f_pos；它们也应该使用 f_pos 字段定位要传送的数据。如第 3 章的“ 读和写 ”一节中介绍的，scull 的实现包

括了这些功能。
scull 处理一个精确定义的数据区，上面的实现对于 scull 来说是有意义的，但大部分设备仅提供了数据流而不是数据区（想想串口和键盘），定位时没有意义的。如果是这种情况，你并不能从声明 lseek 操作中摆脱出来，因为默认方法时允许定位的。相反，你应该使用如下这段代码：
（代码）
刚刚看到的代码摘自 scullpipe 设备，它时不可定位的；尽管错误码的符号名代表“ 是管道 ”，它实际翻译为 “ 非法定位 ”。由于位置指示器 filp->f_pos 对于非可定位设备时没有意义的，在数据传递过程中 read 或 write 都不必更新这个字段。

设备文件的访问控制

有时提供访问控制对于设备节点的可靠性来说时至关重要的。不仅未授权的用户不允许使用设备（可以通过文件系统的权限位来指定），而且有时只允许一次一个授权用户打开设备。
上面给出的代码中除了文件系统权限位外，没有实现任何访问控制。如果 open 系统调用将请求转发给驱动程序，open 就会成功。这里我将介绍实现某些附加检查的新技术。
这里的问题与 tty 的使用很相似。在 tty 的使用中，当用户登录到系统后，login 进程修改设备节点的属主，防止其他进程侵入 tty 数据流。然而，仅仅为了保证对设备的唯一使用，每次都通过特权程序修改设备属主时不现实的。
出现在本节的设备与基本 scull 设备有类似的功能（即，都实现了一个持久性内存区）；它与 scull 的区别仅仅时访问控制，这些控制是在 open 和 close 操作中实现的。

独享设备

提供访问控制的最残忍的方法就是，每次只允许一个进程打开设备（独享）。我个人不喜欢这种技术，因为它阻碍了用户的灵活性。一个用户可能需要在同一个设备上运行多个进程，一个读状态信息，其他进程写数据。通常有许多程序和外壳脚本可以完成许多任务。换句话说，独享更象时策略而非机制（至少我认为这样）。
除了我讨厌独享外，它对于设备驱动程序来说非常易于实现，这里给出代码。源码摘自一个称为 scullsingle 的设备。
open 调用基于一个全局整数标志拒绝访问：
（代码）
另一方面，close 调用标记设备不在忙。
（代码）
存放打开标志（scull_s_count）最佳场所是设备结构（这里是 Scull_Dev），因为从概念上说，它从属于设备。
然而，scull 驱动程序单独使用一个变量保存打开标志，主要时为了与基本 scull 设备使用相同的设备结构，减少代码的重复。

限制每次只由一个用户访问

更有用的访问控制实现是，如果没有其他人能够控制设备，只允许一个使用有访问权。这种

检查时在正常的权限检查后进行的，它可以提供比指定属主和组权限位更严格的访问控制。这种方法与 tty 使用的策略时相同的，但它没有使用一个外部特权程序。
这一功能比起实现独享来要有一定的技巧。此时，需要两个数据项，一个打开计数和设备“ 属主 ”的 UID。再次说明，存放这些数据项的最佳位置是在设备结构内部；但基于和前面 scullsingle 中同样的原因，例子中使用了全局变量。设备的名字时 sculluid。
open 调用在首次打开时授权，但它记下设备的属主。这意味着，用户可以多次打开设备，允许合作进程无缝衔接在一起。与此同时，其他用户不能打开设备，避免了外部干扰。由于这个函数的版本基本和前一个相同，这里仅列出了相关部分：
（代码）
这里虽然代码完成了权限检查，我还是决定返回-EBUSY 而不是-EPERM 来告诉被拒绝的用户正确的原因。返回“ 权限拒绝 ”通常都是检查/dev 文件的权限和属主的结果，而“ 设备忙 ”则正确地告诉用户，已有进程正在使用设备。
由于 close 仅仅时对使用计数减 1，它的代码这里没有列出。

阻塞型打开替代 EBUSY

当设备不能访问时返回一个错误，这是通常最合理的方式，但有些情况下，你最好让进程等待设备。
例如，如果数据通信通道同时用来定时传送报告（使用 crontab），同时也偶尔根据人们的需要使用，定时报告最好稍微延迟一会儿，而不是因为通道正忙就失败返回。
这是程序员在实际驱动程序时必须做出的选择之一，正确的答案依赖所解决的问题。
正如你所猜到的，可以用返回 EBUSY 替代阻塞型打开。
scullwuid 时 sculluid 的另一个版本，它在 open 中等待设备而不是返回-EBUSY。它和 sculluid 的唯一不同时 open 操作中的部分代码：
（代码）
然后，release 方法要负责唤醒任何等待的进程：
（代码）
阻塞型打开实现的问题在于，对于交互式用户来说这是非常不愉快的，他可能会猜想设备出了什么毛病。交互式用户通常使用 cp 和 tar 之类的命令，它们都没有在 open 调用中加上 O_NONBLOCK 选项。隔壁某些用磁带做备份的人更愿意被告知 “ 设备或资源忙 ”，而不是坐在一边，猜想为什么今天使用 tar 时硬盘会不声不响呢。
这类问题（相同设备的不兼容策略）最好为每种访问策略实现一个设备节点的方法来解决，这与/dev/ttyS0 和/dev/cua0 用不同方法操作同一个串口，或是/dev/sculluid 和/dev/scullwuid 提供两种不同策略访问同一内存区很相似。

在打开时克隆设备

管理访问控制的另一项技术是，在进程打开设备时创建设备的一个私有副本。
很明显，如果设备没有绑定到某个硬件对象上的话，这样做是有可能的；scull 就是这种“ 软件 ”设备的例子。kmouse 模块也使用了这一技术，所以每个虚拟控制台好象都有一个鼠标设备。当设备的副本是由软件驱动程序创建的时，我称我们为“ 虚拟设备 ”——就如同“ 虚拟控制台 ”都使用同一个物理 tty 设备一样。
虽然这种对访问控制的需要并不常见，这个实现同样很能说明，内核代码可以轻松地修改从

应用角度看的外部世界（即，计算机）。事实上，这个话题确实有点怪异，如果你不感兴趣，你可以直接跳到下一章。
软件包 scull 中的/dev/scullpriv 设备节点实现了虚拟设备。scullpriv 实现利用进程控制终端的次设备号做为访问虚拟设备的键值；这种选择造成一个完成不同的策略。例如，如果使用 uid，就会造成每个用户一个不同的虚拟设备，而使用 pid 则为每个访问设备的进程创建一个新设备。
决定使用控制终端可以方便地通过输入/输出重定向测试设备。
open 方法如下所示。它必须查找一个合适的虚拟设备，可能的话还要创建一个。由于函数的最后一部分是从基本 scull 中复制过来的，我们在前面已经看过了，这里没有列出。
（代码）
close 方法没有什么特别处理。它在最后一次关闭时释放设备，为了简化对设备的测试，我没有维护打开计数。如果设备在最后一个关闭时释放了，除非有一个后台进程至少打开设备一次，否则在写设备后无法再从设备中读到相同的数据。驱动程序样例选择使用较为简单的方法保存数据，所以在下次打开设备时，你可以找到那些数据。当 cleanup_module 被调用时，释放设备。
这里时/dev/scullpriv 的 close 实现，它也结束了本章。
（代码）

快速索引

本章介绍了如下这些符号和头文件：
#include <linux/ioctl.h>
定义了所有用于定义 ioctl 命令宏的头文件。现在它包含在<linux/fs.h>中。Linux 1.2 没有定义本章介绍那些宏；如果需要向后兼容的话，我建议你看一看 scull/sysdep.h，那里为老版本的内核定义相应的符号。
_IOC_NRBITS
_IOC_TYPEBITS
_IOC_SIZEBITS
_IOC_DIRBITS
ioctl 命令不同位字段的可用位数。还有 4 个宏定义了不同的 MASK（掩码），4 个宏定义了不同的 SHIFT（偏移），但它们基本仅用于内部使用。由于_IOC_SIZEBITS 在不同体系结构上的值不同，它时一个需要检查的比较重要的值。
_IOC_NONE
_IOC_READ
_IOC_WRITE
“ 方向 ”位字段的可能值。“ 读 ”和“ 写 ”时不同的位，可以 OR 在一起实现读/写。这些值都是基于 0 的。
_IOC(dir,type,nr,size)
_IO(type,nr)
_IOR(type,nr,size)
_IOW(tyep,nr,size)
_IOWR(type,nr,size)
用于生成 ioctl 命令的宏。
_IOC_DIR(nr)

_IOC_TYPE(nr)
_IOC_NR(nr)
_IOC_SIZE(nr)
用于解码 ioctl 命令的宏。特别地，_IOC_TYPE(nr)可以是_IOC_READ 和_IOC_WRITE 的 OR 组合。
#include <linux/mm.h>
int verify_area(int mode, const void *ptr, unsigned long extent);
这个函数验证指向用户空间的指针是否可用。verify_area 处理页面失效，必须在除 read 和 write 以外的地方访问用户前调用，read 和 write 被调用时缓冲区已经被验证了。非 0 返回值报告错误，应该将其返回给调用者。在内核 2.1 中使用 verify_area 以及如下这些宏和函数的话，见第 17 章的“访问用户空间”一节。
VERIFY_READ
VERIFY_WRITE
verify_area 中 mode 参数的可能值。VERIFY_WRITE 是 VERIFY_READ 的超集。
#include <asm/segment.h>
void put_user(datum, ptr);
usigned long get_user(ptr);
用从用户空间中存取单个数据的宏。传送的字节个数依赖于 sizeof(*ptr)。版本 1.2 中没有这些函数。如果你想在版本 1.2 和 2.0 中同时编译模块的话，你可以看看 scull/sysdep.h。
void put_user_byte(val, ptr);
usigned char get_user_byte(ptr);
这些函数以及它们的_word 和_long 函数都已经废弃了。2.0 和以后的内核在#ideF WE_REALLY_WANT_TO_USE_A_BROKE_INTERFACE 里定义了这些函数。强烈要求程序员调用 put_user 和 get_user。
#include <linux/sched.h>
void interruptible_sleep_on(struct wait_queue **q);
void sleep_on(struct wait_queue **q);
调用两者其一就可以让当前进程睡眠。通常，你应该选择 interruptible 形式实现阻塞型读和写。
void wake_up(struct wait_queue **q);
void wake_up_interruptible(struct wait_queue **q);
这些函数唤醒睡在队列 q 上的进程。_interruptible 形式的函数仅唤醒可中断的进程。
void schedule(void);
这个从运行队列里挑选一个可运行的进程运行。所选的进程可以时 current 或其他进程。由于 sleep_on 函数在内部调用了 schedule，你通常不会直接调用这个函数。
void select_wait(struct wait_queue **wait_address, select_table *p);
这个将当前进程放到一个等待队列中，但不立即进行调度。它是设计用来实现设备驱动程序的 select 方法的。2.1.23 中完全修改了 select 的实现；详情可见第 17 章的“poll 方法”一节。
#include <linux/fs.h>
SEL_IN
SEL_OUT
SEL_EX
这些宏的某一个做为 mode 参数传递给设备的 select 方法。
int fasync_helper(struct inode *inode, struct file *filp, int mode, struct fasync_struct **fa);

这个函数时实现 fasync 设备方法的“助手”函数。mode 参数与传递给方法的值相同，而 fa 指向一个设备相关的 fasync_struct *变量。

void kill_fasync(struct fasync_struct *fa, int sig);

如果驱动程序支持异步触发，这个函数用来向注册到 fa 的进程发送信号。

第 6 章 时间流

至此，我们知道怎样写一个特性比较完全的字符模块了。我们将在后面几章陆续讨论驱动程序可以访问的一些内核资源。本章，我们先来看看内核代码是如何对时间问题进行处理的。该问题包括(按复杂程度排列)：

- 如何获得当前时间
- 如何将操作延迟指定的一段时间
- 如何调度函数到指定的时间后异步执行

内核中的时间间隔

我们首先要涉及的是时钟中断，操作系统通过时钟中断来确定时间间隔。时钟中断的发生频率设定为 **HZ**，**HZ** 是一个与体系结构无关的常数，在文件<**linux/param.h**>中定义。至少从 2.0 版到 2.1.43 版，Alpha 平台上 Linux 定义 HZ 的值为 1024，而其他平台上定义为 100。

当时钟中断发生时，**jiffies** 值就加 1。因此，**jiffies** 值就是自操作系统启动以来的时钟滴答的数目；**jiffies** 在头文件<**linux/sched.h**>中被定义为数据类型为 **unsigned long volatile** (32 位无符号长整型)的变量，因此连续累加一年又四个多月后就会溢出(假定 **HZ**=100，1 个 **jiffies** 等于 1/100 秒，**jiffies** 可记录的最大秒数为 42949672.96 秒，约合 1.38 年)。如果你打算连续运行 Linux 一年又四个多月以上，你最好买台 Alpha，那么，就是跑五亿年也不会溢出了－Alpha 机器上 **jiffies** 有 64 位。我是无法准确地告诉你 **jiffies** 溢出时会发生些什么的，我可没有那么长的时间来等这件事发生。

如果你修改 **HZ** 值后重编译内核，在用户空间你不会注意到有什么不同。尽管 **jiffies** 值以不同的步长增长，但一切似乎还正常。会产生更多的中断，系统开销更大了，但是因为处理器调度得更频繁了，系统会很不稳定。

我在我的PC上测试了一些**jiffies**值：在 100Hz时，系统的响应很慢；100Hz是缺省值；在 1kHz时，系统跑的相当慢，但响应得很快；在 10kHz时，系统极慢；在 50kHz时，系统已经令人无法忍受了。修改中断频率还有副作用，**jiffies**值溢出要花的时间不同了(10kHz的时钟频率下，只要五天)，BogoMips值的计算精度也不同了[*]。而且还有一些别处都没提及的硬件上的限制。例如，19 是PC上时钟频率的能设的最小值，其他体系结构上也存在着类似的限制。

此外，在使用模块时还要小心。如果你改变了 **HZ** 的定义，你*必须*重新编译和安装你使用的所有模块。内核中一切都与 **HZ** 值有关，包括模块。我是在增加了 **HZ** 值因而无法双击

[*] 由于中断的开销，时钟频率越高，精度就越差。

鼠标后，发现到这一点的。

总而言之，时钟中断的最好方法是保留 **HZ** 的缺省值，因为我们可以完全相信内核的开发者们，他们一定已经为我们挑选了最佳值。关于本专题的更多信息可参见头文件 **<linux/timex.h>**。

获取当前时间

内核一般通过 **jiffies** 值来获取当前时间。尽管该数值表示的是自上次系统启动到当前的时间间隔，但因为驱动程序的生命期只限于系统的运行期(uptime)，所以也是可行的。驱动程序利用 **jiffies** 的当前值来计算不同事件间的时间间隔(我在 *kmouse* 模块中就用它来分辨鼠标的单双击)。简而言之，利用 **jiffies** 值来测量时间间隔还是很有效的。

驱动程序一般不需要知道墙上时间，通常只有象 *cron* 和 *at* 这样用户程序才需要墙上时间。需要墙上时间的情形是使用设备驱动程序的特殊情况，此时可以通过用户程序来将墙上时间转换成系统时钟。

如果驱动程序真的需要获取当前时间，可以使用 *do_gettimeofday* 函数。该函数并不返回今天是本周的星期几或类似的信息；它是用微秒值来填充一个指向 **struct timeval** 的指针变量。相应的原型如下：

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);
```

源码中声明的 *do_gettimeofday* 在 Alpha 和 Sparc 之外的体系结构上有“ 接近微秒级的分辨率 ” ，在 Alpha 和 Sparc 上和 **jiffies** 值的分辨率一样。Sparc 的移植版本在 2.1.34 版的内核中升级了，可以支持更细粒度的时间度量。当前时间也可以通过 **xtime** 变量(类型为 **struct timeval**)获得(但精度差些)；但是，并不鼓励直接使用该变量，因为除非关闭中断，无法原子性地访问 **timeval** 变量的两个域 **tv_sec** 和 **tv_usec**。使用 *do_gettimeofday* 填充的 **timeval** 结构变量会更快些。

令人遗憾的是，1.2 版的 Linux 并未开放 *do_gettimeofday* 函数。如果你要获取当前时间，又希望程序能够向后兼容，你应该使用该函数下面的这个版本：

```
#if LINUX_VERSION_CODE < VERSION_CODE(1,3,46)
/*
 * 内核头文件已经该函数是非静态的。
 * 我们应先用其他名字来实现它，再 #define 它。
 */
extern inline void redo_gettimeofday(struct timeval *tv)
{
 unsigned long flags;

 save_flags(flags);
```

```
 cli();
 *tv=xtime;
 restore_flags(flags);
}
#define do_gettimeofday(tv) redo_gettimeofday(tv)
#endif
```

这个版本比实际的版本精度要差些，因为它只使用 **xtime** 结构的当前值，这个值并不会比 **jiffies** 值的粒度更细。但是，它却能在不同的 Linux 平台间移植。“ 实际的 ” 函数是通过与体系结构相关的代码查询硬件时钟来获得更高的分辨率。

获取当前时间的代码可见于 *jit*("Just In Time")模块，源文件可以从 O'Reilly 公司的 FTP 站点获得。*jit* 模块将创建*/proc/currentime* 文件，它以 ASCII 码的形式返回它读该文件的时间。我选择用动态的*/proc* 文件，是因为这样模块代码量会小些－不值得为返回两行文本而写一整个设备驱动程序。

如果你用 *cat* 命令在一个时钟滴答内多次读该文件，就会发现 **xtime** 和 do_gettimeofday 两者的差异了：

```
morgana%cat /proc/currentime /proc/currentime /proc/currentime
gettime: 846157215.937221
xtime:   846157215.931188
jiffies:   1308094
gettime: 846157215.939950
xtime:   846157215.931188
jiffies:   1308094
gettime: 846157215.942465
xtime:   846157215.941188
jiffies:   1308095
```

延迟执行

使用定时器中断和 **jiffies** 值，时钟滴答的整数倍的时间间隔很容易获得，但更小的时延，程序员必须通过软件循环来获得，这将在本节稍后处介绍。

尽管我将会介绍一些很奇特的技术，但我认为最好先看些简单的延迟实现代码，尽管下面要介绍的第一种实现并不是最好的。

长延迟

如果你想将执行延迟几个时钟滴答或者你对延迟的精度要求不高(比如，你想延迟整数数目的秒数)，最简单的实现(最笨的)如下，也就是*忙等待*：

```
unsigned long j=jiffies+jit_delay*HZ;

while (jiffies<j)
  /* 什么也不做 */
```

这种实现当然要避免*。我在这提到它只是因为有时你可以运行这段代码来更好地理解其他实现(在本章稍后处我会说明如何利用忙等待来做测试)。

还是先看看这段代码是如何工作的。因为内核的头文件中 **jiffies** 被声明为 **volatile** 型变量，每次 C 代码访问它时都会重新读取它，因此该循环可以起到延迟的作用。尽管也是“ 正确 ”的实现，但这个忙等待循环在延迟期间会锁住整台计算机；因为调度器不会中断运行在内核空间的进程。而且当前的内核实现为不可重入的，因此内核中的忙等待循环将会锁住一台 SMP 机器的所有处理器。

更糟糕的是，如果在进入循环之前关闭了中断，**jiffies** 值就不会得到更新，那么 while 循环的条件就永真。你将不得不按下大红按钮(指冷启动)。

这种延迟和下面的几种延迟方法都在 *jit* 模块中实现了。由该模块创建的所有*/proc/jit** 文件每次被读时都延迟整整 1 秒。如果你想测试忙等待代码，可以读*/proc/jitbusy* 文件，当该文件的 *read* 方法被调用时它将进入忙等待循环，延迟 1 秒；而象 *dd if=/proc/jitbusy bs=1* 这样的命令每次读一个字符就要延迟 1 秒。

可以想见，读*/proc/jitbusy* 文件会大大影响系统性能，因为此时计算机要到 1 秒后才能运行其他进程。

更好的延迟方法如下，它允许其他进程在延迟的时间间隔内运行，尽管这种方法不能用于实时任务或者其他时间要求很严格的场合：

```
while (jiffies<j)
   schedule();
```

这个例子和下面各例中的变量 j 应是延迟到达时的 **jiffies** 值，在忙等待时一般就是象这样使用的。

这种循环(可以通过读*/proc/jitsched* 文件来测试它)延迟方法还不是最优的。系统可以调度其他任务；当前任务除了释放 CPU 之外不做任何工作，但是它仍在任务队列中。如果它是系统中唯一的可运行的进程，它还会被运行(系统调用调度器，调度器还是同一个进程，此进程又再调用调度器，然后...)。换句话说，机器的负载(系统中运行的进程个数)至少为 1，而 idle 空闲进程(进程为 0，历史性的被称为"swapper")绝不会被运行。尽管这个问题看来无所谓，当系统空闲时运行 idle 空闲进程可以减轻处理器负载，降低处理器温度，延长处理器寿命，如果是手提电脑，电池的寿命也可延长。而且，延迟期间实际上进程是在执行的，因此这段延迟还是记在它的运行时间上的。运行命令 *time cat /proc/jitsched* 就可以发现到这一点。

尽管有些毛病，这种循环延迟还是提供了一种有点“脏”但比较快的监控驱动程序工作的途径。如果模块中的臭虫(bug)会锁死整个系统，在每个用于调试的 *printk* 语句后都添加一小段延迟，可以保证在处理器碰到令人厌恶的臭虫被锁死之前，所有的打印消息都能进入系统日志(system log)中。如果没有这样的延迟，这些消息能进入内存缓冲区，但在 *klogd* 得到运行前系统可能已经被锁住了。

还有其他更好的获得延迟的方法。在内核态下让进程进入睡眠态的正确方式是设置 **current->timeout** 后睡眠在一个等待队列上。调度器每次运行时都会比较进程的 **timeout** 值和当前的 **jiffies** 值，如果 **timeout** 值小于等于当前时间，那么不管它的等待队列如何进程都会被唤醒。只要没有系统事件唤醒进程使它离开等待队列，那么一旦当前时间达到 timeout 值，调度器就唤醒睡眠进程。

这种延迟实现如下：

```
struct wait_queue *wait=NULL;

current->timeout=j;
interruptible_sleep_on(&wait);
```

注意要调用 *interruptible_sleep_on* 而不是 *sleep_on*，因为调度器不检查不可中断的进程的 **timeout** 值－这种进程的睡眠即使超时也不被中断。因此，如果你调用 *sleep_on*，就无法中断该睡眠进程。你可以通过读*/proc/jitqueue* 文件来测试上面的代码。

Timeout 域是个很有意思的系统资源。它可以用来实现阻塞的系统调用和计算延迟。如果硬件保证只要不出错就能在确定的时间内给出响应，那么驱动程序就可以在恰当设置 **timeout** 值后进入睡眠。例如，如果你有一个对海量存储的数据传输请求(读或者写)，而磁盘响应该请求比如说要 1 秒。如果你设置了 timeout 值，并且当前时间达到它了，进程于是被唤醒，驱动程序开始处理这个请求。如果你使用这种技术，在进程被正常唤醒之后 timeout 值应被清为零。而如果进程是因为 **timeout** 超时而被唤醒的，调度器会清这个域，驱动程序就不必再做了。

你可能注意到了，如果目的只是插入延迟，这里并没有必要使用等待队列。实际上，如下所示，用 **current->timeout** 而不用等待队列就可以达到目的：

```
current->timeout=j;
current->state=TASK_INTERRUPTIBLE;
schedule();
current->timeout=0;/* 重置 timeout 值*/
```

这段语句是在调用调度器之前先改变进程的状态。进程的状态被标记为 **TASK_INTERRUPTIBLE**(与 **TASk_RUNNING** 相对应)，这保证了该进程在超时前不会被再次运行(但其他系统事件如信号可能会唤醒它)。这种延迟方法在文件*/proc/jitself* 中实现了－这个名字强调了，读进程是“自己进入睡眠的”，而不是通过调用 *sleep_on*。

---

* 尤其在SMP机器上要避免，这种实现可能会锁住整个机器。

短延迟

有时驱动程序需要非常短的延迟来和硬件同步。此时，使用 **jiffies** 值就不能达到目的。

这时就要用内核函数*udelay*[*]。它的原型如下：

```
#include <linux/delay.h>
void udelay(unsigned long usecs);
```

该函数在绝大多数体系结构上是作为内联函数编译的，并且使用软件循环将执行延迟指定数量的微秒数。这里要用到 BogoMips 值：*udelay* 利用了整数值 **loops_per_second**，这个值是在启动时计算 BogoMips 时得到的。

udelay 函数只能用于获取较短的时间延迟，因为 **loops_per_second** 值的精度就只有 8 位，所以当计算更长的延迟时会积累下相当大的误差。尽管运行的最大延迟将近 1 秒(因为更长的延迟就要溢出)，推荐的 *udelay* 函数的参数的最大值是取 1000 微秒(1 毫秒)。

要特别注意的是 *udelay* 是个忙等待函数，在延迟的时间段内无法运行其他的任务。源码见头文件**<asm/delay.h>**。

目前内核不支持大于 1 微秒而小于 1 个时钟滴答的延迟，但这不是个问题，因为延迟是给硬件或者人去识别的。百分之一秒的时间间隔对人来说延迟精度足够了，而 1 毫秒对硬件来说延迟时间也足够长。如果你*真的*需要其间的延迟间隔，你只要建立一个连续执行*udelay(1000)*函数的循环。

任务队列

许多驱动程序需要将任务延迟到以后处理，但又不想占用中断。Linux 为此提供了两种方法：任务队列和内核定时器。任务队列的使用很灵活，可以或长或短地延迟任务到以后处理，在写中断处理程序时任务队列非常有用，在第 9 章“ *中断处理* ”中，我们还将在“ 下半部处理 ”一节中继续讨论。内核定时器则用来调度任务在未来某个相对精确的时间执行，将在本章的“ 内核定时器 ”一节中讨论。

要使用到任务队列的一个典型情形是，硬件不产生中断，但仍希望提供阻塞的读。此时需要对设备进行轮询，但要小心地不使 CPU 负担过多无谓的操作。将读进程到指定的时间后(例如，使用 **current->timeout** 变量)唤醒并不是个很好的方法，因为每次轮询需要两次上下文切换，而且通常轮询机制在进程上下文之外才可能较好地实现。

类似的情形还有象不时地给简单的硬件设备提供输入。例如，有一个直接连接到并口的步进马达，要求该马达能一步步地移动。在这种情况下，由控制进程通知设备驱动程序进行移动，但实际上移动是在 *write* 返回后才一步步地进行的。

[*] u表示希腊字母"mu"(μ)，它代表“ 微 ”。

快速完成这类不固定的任务的恰当方法是注册任务在未来执行。内核提供了对任务“队列”的支持，任务可以累积到队列上一块“运行”。你可以声明你自己的任务队列并且随意地操纵它，或者也可以将你的任务注册到预定义的任务队列中去，由内核来运行它。

下面一节将先概述任务队列，然后介绍预定义的任务队列，这让你可以开始进行一些有趣的测试(如果出错也可能挂起系统)，最后介绍如何运行你自己的任务队列。

任务队列的特性

任务队列是任务的一张列表，每个任务用一个函数指针和一个参数表示。任务运行时，它接受一个 **void *类型的参数，返回值类型为 **void**。而参数指针 data 可用来将一个数据结构传入函数，或者可以被忽略。队列本身是结构(任务)的列表，为声明和操纵它们的内核模块所拥有。这些模块全权负责这些数据结构的分配和释放；为此一般使用静态的数据结构。

队列元素由下面这个结构来描述，这段代码是直接从头文件**<linux/tqueue.h>**拷贝下来的：

```
struct tq_struct {
        struct tq_struct *next;          /* 激活的 bh 的链接表 */
        unsigned long sync;                 /* 必须初始化为零 */
        void (*routine)(void *);          /* 调用的函数 */
        void *data;                          /* 传递给函数的参数 */
};
```

第一行注释中的 bh 指的是*下半部处理程序(bottom-half)*。下半部处理程序是“中断处理处理程序的下半部”；我们将在第 9 章的“下半部处理程序”一节介绍中断时详细讨论。

任务队列是处理异步事件的重要资源，而且绝大多数的中断处理程序将它们的任务延迟到任务队列被处理时执行。另外，有些任务队列*是*下半部处理程序，通过调用 *do_bottom_half* 函数来处理。本章并不要求你理解下半部处理，但必要时我也会涉及到。

上面的数据结构中最重要的字段是 **routine** 和 **data**。将要延迟的任务插入队列，必须先设置好结构的这些字段，并把 **next** 和 **sync** 两个字段清零。结构中的 **sync** 标志位用于避免同一任务被插入多次，这会破坏 **next** 指针。一旦任务被排入队列，该数据结构就被认为为内核“拥有”了，不能再被修改。

与任务队列有关的其他数据结构还有 **task_queue**，目前它实现为指向 **tq_struct** 结构的指针；如果将来需要扩充 **task_queue**，只要用 **typedef** 将该指针定义为其他符号就可以了。

下面的列表汇总了所有可以对 **tq_struct** 结构进行的操作；所有的函数都是内联的。

void queue_task(struct tq_struct *task, task_queue *list);

正如该函数的名字，本函数用于将任务排进队列中。它关闭了中断，避免了竞争，因此

可以被模块中任一函数调用。

void queue_task_irq(struct tq_struct *task, task_queue *list);

与前者类似，但本函数只能由不可重入的函数调用(象中断处理程序，所以本函数的名字带上了 irq)。它比 *queue_task* 函数要快一些，因为它在排队前不关闭中断。如果你在一个可重入的函数内调用本函数，由于没有屏蔽资源竞争，是很危险的。但是，本函数排除了“ 运行时排队 ”的情形(也即将任务插入正在运行的那个任务的位置上)。

void queue_task_irq_off(struct tq_struct *task, task_queue *list);

本函数只能在中断已关闭的情况下调用。它比前两个函数要快，但没有防止象“ 并发排队 ”和“ 运行时排队 ”这样的资源竞争。

void run_task_queue(struct tq_struct *task, task_queue *list);

run_task_queue 函数用于运行累积在队列上的任务。除非你要声明和维护自己的任务队列，否则不必调用本函数。

2.1.30 版的内核已经不提供 *queue_task_irq* 和 *queue_task_irq_off* 这两个函数了，被认为得不偿失。详情见第 17 章“ *最近的发展* ” 的“ 任务队列 ”一节。

在研讨任务队列的细节之前，最好还是先介绍一下内部的一些实现细节。任务队列与相应的系统调用是异步执行的；这种异步执行特别需要注意，必须先介绍一下。

任务队列要在*安全的时间内*运行。这里安全的意思是在执行时没有什么特别严格的要求。因为在处理任务队列时允许硬件中断，任务代码也不要求执行的非常快。但队列中的函数执行得也不能太慢，毕竟在整个处理任务队列的期间，只有硬件中断才能被系统处理。

另一个与任务队列有关的概念是*中断时间*。在Linux中，中断时间是个软件上的概念，取决于内核的全局变量**intr_count**。任一时候该变量都记录了正在执行的中断处理程序被嵌套的层数*。

一般的计算流程中，当处理器允许某个进程时，**intr_count** 值为 0。当 **intr_count** 不为零时，，执行的代码就与系统的其他部分是异步的了。这些异步代码可以是硬件中断的处理或者是“ 软件中断 ”－与任何进程都无关的一个任务，我们称它在“ 中断时间内 ”运行。这种异步代码是不允许做某些操作的；特别的，它不能使当前进程进入睡眠，因为 current 指针的值与正在运行的软件中断代码无关。

典型的例子是退出系统调用时要执行的代码。如果因为某个原因此时还有任务需要得到执行，内核可以一退出系统调用就处理它。这是个“ 软件中断 ”，**intr_count** 值在处理这个待执行的任务之前会先加 1。由于主线指令流被中断了，该函数算是在“ 中断时间 ”内被处理的。

当 **intr_count** 非零时，不能激活调度器。这也就意味着不允许调用 **kmalloc(GFP_KERNEL)**。在中断时间内，只能进行原子性的分配(见第 7 章“ *掌握内存* ”

的“优先权参数”一节)，而原子性的分配较“普通的”分配更容易失败。

如果运行在中断时间的代码调用了调度器，类似“Aiee: scheduling in interrupt”这样的错误信息和以 16 进制显示的调用点处的地址会打印到控制台上。2.1.37 之后的版本，oops 消息也会打印出来，通过分析寄存器的值可以进行调试。在中断时间内如果试图非原子性地按优先权分配内存，也会显示包括着调用者的调用点处地址的错误信息。

预定义的任务队列

延迟任务执行的简单方法是使用内核维护的任务队列。这种队列有下面描述的四种，但驱动程序只能用前三种。任务队列的定义在头文件<**linux/queue.h**>中，你的驱动程序代码要将它包含(include)进来。

tq_scheduler 队列

当调度器被运行时该队列就会被处理。因为此时调度器在被调度出的进程的上下文中运行，所以该队列中的任务几乎可以做任何事；它们不会在中断时运行。

tq_timer 队列

该队列由定时器队列处理程序(timer tick)运行。因为该处理程序(见函数 *do_timer*)是在中断时间运行的，该队列中的所有任务就也是在中断时间内运行的了。

tq_immediate 队列

立即队列在系统调用返回时或调度器运行时尽快得到处理的(不管两种情况谁先发生了)。该队列是在中断时间内得到处理的。

tq_disk 队列

1.2 版的内核不再提供这种任务队列了，内存管理例程内部使用，模块不能使用。

使用任务队列的一个设备驱动程序的执行流程可见图 6-1。该图演示了设备驱动程序是如何在中断处理程序中将一个函数插入 **tq_scheduler** 队列中的。

被执行的代码
中断
从中断返回
数据
关键字
处理器代码
调度器
驱动程序代码
(指向任务)
"sync"位
(指向 next)
图 6-1:任务队列使用的执行流程

[*] 2.1.34 版的内核不再使用**intr_count**变量。详情见第 17 章的“中断管理”一节。

示例程序是如何工作的

延迟计算的示例程序是 *jiq*(Just In Queue)模块，本节中抽取了它的部分源码。该模块创建*/proc* 文件，可以用 *dd* 或者其他工具来读；这与 *jit* 模块很相似。该示例程序使用了动态*/proc* 文件因此不能在 Linux1.2 上运行。

读*jiq*文件的进程进入睡眠状态直到缓冲区满*。缓冲区由不断运行的任务队列来填充。任务队列的每遍运行都将在要填充的缓冲区中添加一个字符串；该字符串记录了当前时间(jiffies值)，该遍的current进程和intr_count值。

该/proc 文件最好是用 dd count=1 命令一次性地读进来；如果你用 *cat* 命令，*read* 方法要被多次调用，输出结果会有重迭，详情可见第 4 章“ *调试技术* ”的“ 使用/proc 文件系统 ”一节。

填充缓冲区的代码都在 *jiq_print* 函数中，任务队列的每遍运行都要调用它。打印函数没什么意思，不在这里列出；我们还是来看看插入队列的任务的初始化代码：

```
struct tq_struct jiq_task; /*  全局变量；初始化为零  */

/*  该行在 init_module()中  */
jiq_task.routine = jiq_print;
jiq_task.data = (void *)&jiq_data;
```

这里没必要清零 **jiq_task** 结构变量的 **sync** 域和 **next** 域，因为静态变量已由编译器初始化为零了。

调度器队列

最容易使用的任务队列是 **tq_scheduler** 队列，因为该队列中的任务不会在中断时间内运行，因此少了很多限制。

/proc/jiqsched 文件是使用 **tq_scheduler** 队列的示例文件。该文件的 *read* 函数以如下的方式将任务 jiq_task 放进 **tq_scheduler** 队列中：

```
/*
 *  使用调度器队列的例子    -- /proc/jiqsched
 */
int jiq_read_sched(char *buf, char **start, off_t offset, int len, int unused)
{
      jiq_data.len = 0;                   /*  还未打印，长度为 0 */
      jiq_data.buf = buf;                 /*  打印到这个缓冲区中  */
```

* */proc*文件的缓冲区是内存中的一页：4KB或 8KB。

```
    jiq_data.jiffies = jiffies;         /*  开始时间  */

    /* jiq_print 会调用  queue_task()  使自己重新进入  jiq_data.queue 队列  */
    jiq_data.queue = &tq_scheduler;

    queue_task(&jiq_task, &tq_scheduler); /*  准备运行*/
    interruptible_sleep_on(&jiq_wait);        /*  进入睡眠队列只到任务完成  */

    return jiq_data.len;
}
```

读读*/proc/jiqsched* 文件很有意思，因为它显示调度器在何时运行－**jiffies** 值表明调度器激活的时间。如果系统中有些正在占用 CPU 的进程，那么队列中各任务的运行间会有些延迟；因为调度器要在若干时钟滴答后才会抢先那些进程。打开这个文件会花上好几秒钟，因为它长达 100 行(在 Alpha 机器上是 200 行)。

测试这些情形最简单方法是跑一个执行空循环的进程。*load50* 是个增加机器负载的程序，它在用户空间执行 50 个并发的忙循环；你可以在示例程序中找到它的源码(*misc-progs/load50.c*)。当在系统中运行 *load50* 程序时，*head* 命令将从*/proc/jiqsched* 文件中抽取类似如下的信息：

```
time      delta       intr_count pid command
1643733     0             0       701 head
1643747    14             0       658 load50
1643747     0             0         3 kswapd
1643755     8             0       655 load50
1643761     6             0       666 load50
1643764     3             0       650 load50
1643767     3             0       661 load50
1643769     2             0       659 load50
1643769     0             0         6 loadmonitor
```

注意到调度队列是在进入 *schedule* 过程后就执行的，因此 **current** 进程就是刚刚被调度出去的进程。这就是为什么*/proc/jiqsched* 文件的第一行总是读该文件的那个进程；它正进入睡眠状态，就要被调出。还可以发现，*kswapd* 和 *loadmonitor*(这是我在我的系统上运行的一个程序)的执行时间都少于 1 个时钟滴答，而 *load50* 是在它的时间片耗尽后被抢先，这离它获得处理器有好几个时钟滴答。

当系统中实际上没有任何进程在运行时，current 进程总是空闲(idle)任务(0 号进程，历史性的被称为"swapper")，任务队列或者是不断地运行或者是每隔 1 个时钟滴答运行一次。如果处理器不能进入“ 暂停 ”("halted")状态，调度器和任务队列就将不断运行；如果处理器能被 0 号进程暂停(halt)，它们每隔 1 个时钟滴答才运行一次。暂停的(halted)处理器只能由中断唤醒。当中断发生时，空闲进程运行调度器(及相应的队列)。下面显示了在没有负载的系统运行运行命令 *head /proc/jiqsched* 得到的结果：

time	delta	intr_count	pid	command
1704475	0	0	730	head
1704476	1	0	0	swapper
1704477	1	0	0	swapper
1704478	1	0	0	swapper
1704478	0	0	6	loadmonitor
1704479	1	0	0	swapper
1704480	1	0	0	swapper
1704481	1	0	0	swapper
1704482	1	0	0	swapper

定时器队列

定时器队列的使用方法和调度器队列差不多。和调度器队列不同的是，定时器队列是在中断时间内执行的。另外，该队列一定在下一个时钟滴答被运行，因此就与系统负载无关了。下面是在我的系统在跑编译程序时运行命令 *head /proc/jiqtimer* 输出的结果：

time	delta	intr_count	pid	command
1760712	1	1	945	cc1
1760713	1	1	945	cc1
1760714	1	1	945	cc1
1760715	1	1	946	as
1760716	1	1	946	as
1760717	1	1	946	as
1760718	1	1	946	as
1760719	1	1	946	as
1760720	1	1	946	as

当前的任务队列实现有一个特性，一个任务可以将自己重新插回到它原先所在的队列。例如，定时器队列中的任务可以在运行时将自己插回到定时器队列中去，从而在下一个时钟滴答又再次被运行。在处理任务队列之前，由于是先用 **NULL** 指针替换队列的头指针，因此才可能进行不断的重新调度。这种实现可追溯到 1.3.70 版的内核。在早期的版本中(象 1.2.13 版)，重调度是不可能的，因为内核在运行队列前不会先整理它。在 1.2 版的 Linux 中试图重新调度任务会因进入死循环(tight loop)而挂起系统。是否具有重新调度的能力是任务队列实现上 1.2.13 版和 2.0.x 各版本间唯一的一处差别。

尽管一遍遍地重新调度同一个任务看起来似乎没什么意义，但有时这也有些用处。例如，我的计算机就是在到达目的地之前让任务在定时器队列上不断地重新调度自己，来实现步进马达的一步步移动的。其他的例子还有 *jiq* 模块，该模块中的打印函数都又重新调度了自己来显示对任务队列一遍扫描的结果。

立即队列

最后一个可由模块代码使用的预定义队列是立即队列。它和下半部中断处理程序工作机制相似，因此必须用 **mark_bh(IMMEDIATE_BH)**标记该处理程序是活跃的。出于高效的目的，只有被标记为活跃的下半部处理程序才会被执行。注意必须在调用 *queue_task 后才*能标记下半部处理程序，否则会带来竞争。详情见第 9 章的“ 下半部处理 ”一节。

立即队列是系统处理得最快的队列－在 **intr_count** 变量加 1 之后马上执行。该队列执行得如此“ 立即 ”以致于当你重新注册你的任务之后，它一返回就立即重新运行。该队列一遍遍执行直到队列为空。只要看看*/proc/jiqimmed* 文件，就会明白它执行得如此只快的原因，在整个读过程中它完全控制了 CPU。

立即队列是由调度器执行的或者是在一个进程从系统调用返回时被执行的。值得注意的是，调度器(至少对于 2.0 版的内核)并不总会一直处理立即队列到它为空；只有从系统调用返回时才会这么做。这可以从下面的示例输出看出来－只有 *jiqimmed* 文件的第一行是当前进程 *head*，而下面各行都不是了。

```
    time      delta      intr_count pid command
 1975640      0              1      1060 head
 1975641      1              1         0 swapper
 1975641      0              1         0 swapper
 1975641      0              1         0 swapper
 1975641      0              1         0 swapper
 1975641      0              1         0 swapper
 1975641      0              1         0 swapper
 1975641      0              1         0 swapper
 1975641      0              1         0 swapper
```

显然该队列不能用于延迟任务的执行－它是个“ 立即 ”队列。相反，它的目的是使任务尽快地得以执行，但是要在“ 安全的时间 ”内。这对中断处理非常有用，因为它提供在实际的中断处理程序之外执行处理程序代码的一处入口。

尽管*/proc/jiqimmed* 将任务重新注册到立即队列中，但这种技术在实际的实现代码中并不鼓励；象这种不肯合作的行为会在整个不断重等记的过程中独占住处理器，那还不如将整个任务一次性地完成。

运行自己定制的工作队列

声明新的任务队列不困难。驱动程序可以随意地声明任意多的新任务队列；这些队列的使用和 **tq_scheduler** 队列差不多。

与预定义队列不同的，内核不会自动处理定制的任务队列。定制的任务队列要由程序员自己维护。

下面的宏声明一个定制队列，在你需要任务队列声明处这个宏会被扩展：

DECLARE_TASK_QUEUE(tq_custom);

声明完队列，就可以调用下面的函数对任务进行排队。上面的宏和下面的调用相匹配：

queue_task(&custom_task,&tq_custom);

然后就可以调用下面的函数运行 **tq_custom** 队列了：

run_task_queue(&tq_custom);

如果现在你想测试你定制的任务队列，你可以在预定义的队列中注册一个函数来处理这个队列。尽管看起来象绕了弯路，但其实并非如此。如果你希望累积任务以便同时得到执行，尽管需要用另一个队列来决定这个“同时”，定制的任务队列还是非常有用的。

内核定时器

内核中最终的计时资源还是定时器。定时器用于调度函数(定时器处理程序)在未来某个特定时间执行。与任务队列不同，你可以指定你的函数在未来*何时*被调用，但你不能确定任务队列中的任务何时执行。另外，内核定时器与任务队列相似的，内核定时器注册的处理函数只执行一次－定时器不是循环执行的。

有时你要执行的操作不在任何进程上下文内，比如关闭软驱马达和中止某个耗时的关闭操作。在这些情况下，延迟从 *close* 调用的返回对应用程序不公平。而且这时也没有必要使用任务队列，因为队中的任务在估算时间的同时还要不断重新注册自己。

这里用定时器就更方便。你注册你的处理函数一次，当定时器超时后内核就调用它一次。这种处理一般较适合由内核完成，但有时驱动程序也需要，就象软驱马达的例子。

Linux 使用了两种定时器，所谓的“旧定时器”和新定时器。在介绍如何使用更好的新定时器前，我先简要介绍一下旧定时器。新定时器，实际上并不新；它们在 1.0 版之前的 Linux 里就已经引入了。

旧定时器包括 32 个静态的定时器。它们的存在只是出于兼容性的考虑(因为替换旧定时器需要修改和测试大量的驱动程序代码)。

旧定时器的数据结构包括一个标明活动的定时器的位屏蔽码和定时器数组，数组的每个成员又包括一个处理程序和该定时器的超时值。旧的定时器结构的主要问题在于，每个需要定时器来延迟操作的设备都要静态地分配给一个定时器。

这种实现在几年前是可以接受的，当时支持的设备(因此需要的定时器)还很有限，但对当前的 Linux 版本就不够了。

我不再介绍如何使用旧的定时器；我在这里提到它们只是为了满足那些好奇的读者。

新的定时器列表

新的定时器被组织成双向链接表。这意味着你加入任意多的定时器。定时器包括它的timeout(超时)值(单位是 jiffies)和超时时调用的函数。定时器处理程序需要一个参数，该参数和处理程序函数指针本身一起存放在一个数据结构中。

定时器列表的数据结构如下，抽取自头文件<**linux/timer.h**>:

```
struct timer_list {
        struct timer_list *next;   /*不要直接修改它 */
        struct timer_list *prev; /*不要直接修改它 */
        unsigned long expires; /* timeout 超时值，以 jiffies 值为单位 */
        unsigned long data; /* 传递给定时器处理程序的参数 */
        void (*function)(unsigned long); /* 超时时调用的定时器处理程序 */
};
```

可以看到，定时器的实现和任务队列有所不同，尽管两个表的表项有些相似。这两个数据结构是由两个编程者几乎在同时创建的，因此不大一样；它们并没有相互复制。所以，定时器处理程序的形参是 **unsigned long** 类型，而不是 **void *** 类型，而定时器处理程序的名字是 function 而非 routine。

定时器的 timeout 值是个"jiffy"值，当 **jiffies** 值大于等于 **timer->expires** 时，就要运行 **timer->function** 函数。Timeout 值是个绝对数值；它与当前时间无关，不需要更新。

一初始化完 **timer_list** 结构，*add_timer* 函数就将它插入一张有序表中，该表每秒钟会被查询 100 次左右(尽管时钟滴答的频率有时要比这高，但这样节省 CPU 时间)。

简单说，操作定时器的有如下函数：

void init_timer(struct timer_list *timer);

该内联函数用来初始化新定时器队列结构。目前，它只将 **prev** 和 **next** 指针清零。建议程序员使用该函数来初始化定时器而不要显式地修改对结构内的指针，以保证向前兼容。

void add_timer(struct timer_list * timer);

该函数将定时器插入挂起的定时器的全局队列。有意思的是，内核定时器最初的实现和现在的实现不同；在 1.2 版的内核中，*add_timer* 函数认为 **timer->expires** 值是相对于当前的 jiffy 值的，所以它在将结构插入全局列表前会先将 jiffies 值加到 timer->expires 上。这个不兼容在示例源代码文件 *sysdep.h* 中得到处理。

int del_timer(struct timer_list *timer);

如果需要在定时器超时前将它从列表中删除，应调用 *del_timer* 函数。但当定时器超时是，系统会自动地将它从列表中删除。

使用定时器的一个例子是 *jiq* 示例模块。*/proc/jittimer* 文件使用一个定时器来产生两行数据；print 函数和前面任务队列用的是同一个。第一行数据是调用 *read* 产生的，而第二行是 100 jiffies 后定时器处理函数打印出的。

/proc/jittimer 文件的代码如下：

```
struct timer_list jiq_timer;

void jiq_timedout(unsigned long ptr)
{
      jiq_print((void *)ptr);                  /*  打印一行数据  */
      wake_up_interruptible(&jiq_wait);    /*  唤醒进程  */
}

int jiq_read_run_timer(char *buf, char **start, off_t offset,
                              int len, int unused)
{

      jiq_data.len = 0;                 /*  准备传递给 jiq_print()函数的各个参数  */
      jiq_data.buf = buf;
      jiq_data.jiffies = jiffies;
      jiq_data.queue = NULL;          /*  不会重新进入队列  */

      init_timer(&jiq_timer);                      /*  初始化定时器结构  */
      jiq_timer.function = jiq_timedout;
      jiq_timer.data = (unsigned long)&jiq_data;
      jiq_timer.expires = jiffies + HZ; /* 1 秒  */

      jiq_print(&jiq_data);      /*  打印并进入睡眠  */
      add_timer(&jiq_timer);
      interruptible_sleep_on(&jiq_wait);
      return jiq_data.len;
}
```

运行命令 **head /proc/jitimer** 得到如下输出结果：

```
   time       delta      intr_count    pid    command
2121704         0           0          1092    head
2121804       100           1             0    swapper
```

很明显地，从第 2 行的 **intr_count** 变量的值可以发现定时器程序是在“ 中断时 ”运行的。

可能看起来有点奇怪的是,定时器总是可以正确地超时,即使处理器正在执行系统调用。我在前面曾提到，运行在内核态的进程不会被调出；但时钟中断是个例外，它与当前进程无关。你可以试试在前台同时读*/proc/jitbusy* 文件和*/proc/jittimer* 文件，这时尽管看起来系统似乎被忙等待的系统调用给锁死住了，但定时器队列和内核定时器还是能不断得到处理的。

快速参考

本章引入如下符号：

#include <linux/param.h>

HZ

HZ 符号指出每秒钟产生的时钟滴答数。

volatile unsigned long jiffies

jiffies 变量每个时钟滴答后加 1；因此它每秒增加 1 个 **HZ**。

#include <linux/time.h>

void do_gettimeofday(struct timeval *tv);

该函数返回当前时间。1.2 版的内核并不提供。

#include <linux/delay.h>

void udelay(unsigned long usecs);

udelay 函数延迟整数数目的微秒数，但不应超过 1 毫秒。

#include <linux/tqueue.h>

void queue_task(struct tq_struct *task, task_queue *list);

void queue_task_irq();

void queue_task_irq_off();

这些函数注册延迟执行的任务。第一个函数，*queue_task*，总是可以被调用；第二个函数只能在不可重入的函数内被调用，而最后一个函数只有在关闭中断后才能被调用。新近的内核只提供第一种函数接口了(见第 17 章的 “ 任务队列 ” 一节)。

void run_task_queue(task_queue *list);

该函数运行任务队列。

task_queue tq_immediate, tq_timer, tq_scheduler;

这些预定义的任务队列在每个时钟滴答后并在内核调度新的进程前尽快地分别得到执行。

#include <linux/timer.h>

void init_timer(struct timer_list *timer);

该函数初始化新分配的定时器队列。

void add_timer(struct timer_list * timer);

该函数将定时器插入待处理的定时器的全局队列。

int del_timer(struct timer_list *timer);

del_timer 函数将定时器从挂起的定时器队列中删除。如果队列中有该定时器，*del_timer* 返回 1，否则返回 0。

第 7 章 获取内存

到目前为止，我们总是用 *kmalloc* 和 *kfree* 来进行内存分配。当然，只用这些函数的确是管理内存的捷径。本章将会介绍其他一些内存分配技术。但我们目前并不关心不同的体系结构实际上是如何进行内存管理的。因为内核为设备驱动程序提供了一致的接口，本章的模块都不必涉及分段，分页等问题。另外，本章我也不会介绍内存管理的内部细节，这些问题将留到第 13 章“ *Mmap 和 DMA* ”的“ Linux 的内存管理 ”一节讨论。

kmalloc 函数的内幕

kmalloc 内存分配引擎功能强大，由于和 *malloc* 函数很相似，很容易就可以学会。这个函数运行得很快－－除非它被阻塞－－它不清零它获得的内存空间；分配给它的区域仍存放着原有的数据。在下面几节，我会详细介绍 *kmalloc* 函数，你可以将它和我后面要介绍的一些内存分配技术作个比较。

优先权参数

kmalloc 函数的第一个参数是 size(大小)，我留在下个小节介绍。第二个参数，是优先权，更有意思，因为它会使得 *kmalloc* 函数在寻找空闲页较困难时改变它的行为。

最常用的优先权是 **GFP_KERNEL**，它的意思是该内存分配(内部是通过调用 *get_free_pages* 来实现的，所以名字中带 **GFP**)是由运行在内核态的进程调用的。也就是说，调用它的函数属于某个进程的，使用 **GFP_KERNEL** 优先权允许 *kmalloc* 函数在系统空闲内存低于水平线 **min_free_pages** 时延迟分配函数的返回。当空闲内存太少时，*kmalloc* 函数会使当前进程进入睡眠，等待空闲页的出现。

新的页面可以通过以下几种途径获得。一种方法是换出其他页；因为对换需要时间，进程会等待它完成，这时内核可以调度执行其他的任务。因此，每个调用 **kmalloc(GFP_KERNEL)**的内核函数都应该是可重入的。关于可重入的更多细节可见第 5 章“ *字符设备驱动程序的扩展操作* ”的“ 编写可重入的代码 ”一节。

并非使用 **GFP_KERNEL** 优先权后一定正确；有时 *kmalloc* 是在进程上下文之外调用的－－比如，在中断处理，任务队列处理和内核定时器处理时发生。这些情况下，**current** 进程就不应该进入睡眠，这时应该就使用优先权 **GFP_ATOMIC**。原子性(atomic)的内存分配允许使用内存的空闲位，而与 **min_free_pages** 值无关。实际上，这个最低水平线值的存在就是为了能满足原子性的请求。但由于内核并不允许通过换出数据或缩减文件系统缓冲区来满足这种分配请求，所以必须还有一些真正可以获得的空闲内存。

为 *kmalloc* 还定义了其他一些优先权，但都不经常使用，其中一些只在内部的内存管理算法中使用。另一个值的注意的优先权是 **GFP_NFS**，它会使得 NFS 文件系统缩减空闲列表

到 **min_free_pages** 值以下。显然，为使驱动程序“ 更快 ”而用 **GFP_NFS** 优先权取代 **GFP_KERNEL** 优先权会降低整个系统的性能。

除了这些常用的优先权，*kmalloc* 还可以识别一个位域：**GFP_DMA**。**GFP_DMA** 标志位要和 **GFP_KERNEL** 和 **GFP_ATOMIC** 优先权一起使用来分配用于直接内存访问(DMA)的内存页。我们将在第 13 章的“ 直接内存访问 ”一节讨论如何使用这个标志位。

size 参数

系统物理内存的管理是由内核负责的，*物理*内存只能按页大小进行分配。这就需要一个面向页的分配技术以取得计算机内存管理上最大的灵活性。类似 *malloc* 函数的简单的线性的分配技术不再有效了；在象 Unix 内核这样的面向页的系统中内存如果是线性分配的就很难维护。空洞的处理很快就会成为一个问题，会导致内存浪费，降低系统的性能。

Linux 是通过维护页面池来处理 *kmalloc* 的分配要求的，这样页面就可以很容易地放进或者取出页面池。为了能够满足超过 **PAGE_SIZE** 字节数大小的内存分配请求，*fs/kmalloc.c* 文件维护页面簇的列表。每个页面簇都存放着连续若干页，可用于 DMA 分配。在这里我不介绍底层的实现细节，因为内部的数据结构可以在不影响分配语义和驱动程序代码的前提下加以改变。事实上，2.1.38 版已经将 kmalloc 重新实现了。2.0 版的内存分配实现代码可以参见文件 *mm/malloc.c*，而新版的实现在文件 *mm/slab.c* 中。想了解 2.0 版实现的详情可参见第 16 章“ *内核代码的物理布局* ”的“ 分配和释放 ”一节。

Linux 所使用的分配策略的最终方案是，内核只能分配一些预定义的固定大小的字节数组。如果你申请任意大小的内存空间，那么很可能系统会多给你一点。

这些预定义的内存大小一般“ 稍小于 2 的某次方 ”(而在更新的实现中系统管理的内存大小恰好为 2 的各次方)。如果你能记住这一点，就可以更有效地使用内存了。例如，如果在 Linux 2.0 上你需要一个 2000 字节左右的缓冲区，你最好还是申请 2000 字节，而不要申请 2048 字节。在低与 2.1.38 版的内核中，申请恰好是 2 的幂次的内存空间是最糟糕的情况了－内核会分配两倍于你申请空间大小的内存给你。这也就是为什么在示例程序 *scull* 中每个单元(quantum)要用 4000 字节而不是 4096 字节的原因了。

你可以从文件 *mm/malloc.c*(或者 *mm/slab.c*)得到预定义的分配块大小的确切数值，但注意这些值可能在以后的版本中被改变。在当前的 2.0 版和 2.1 版的内核中，都可以用个小技巧－尽量分配小于 4K 字节的内存空间，但不能保证这种方法将来也是最优的。

无论如何，Linux2.0 中 *kmalloc* 函数可以分配的内存空间最大不能超过 32 个页－Alpha 上的 256KB 或者 Intel 和其他体系结构上的 128KB。2.1.38 版和更新的内核中这个上限是 128KB。如果你需要更多一些空间，那么有下面一些的更好的解决方法。

get_free_page 和相关函数

如果模块需要分配大块的内存，那使用面向页的分配技术会更好。请求整页还有其他一

些好处，后面第 13 章的“ mmap 设备驱动程序操作 ”一节将会介绍。

分配页面可使用下面一些函数：

- *get_free_page* 返回指向新页面的指针并将页面清零。
- *__get_free_pages* 和 *get_free_page* 类似，但不清零页面。
- *__get_free_pages* 返回一个指向大小为几个页的内存区域的第一个字节位置的指针，但也不清零这段内存区域。
- *__get_dma_pages* 返回一个指向大小为几个页的内存区域的第一个字节位置的指针；这些页面在物理上是连续的，可用于 DMA 传输。

这些函数的原型在 Linux2.0 中定义如下：

```
unsigned long get_free_page(int priority);
unsigned long __get_free_page(int priority);
unsigned long __get_dma_pages(int priority, unsigned long order);
unsigned long __get_free_pages(int priority, unsigned long order, int dma);
```

实际上，除了*__get_free_pages*，这些函数或者是宏或者是最终调用了*__get_free_pages*的内联函数。

当程序使用完分配给它的页面，就应该调用下面的函数。下面的第一个函数是个宏，其中调用了第二个函数：

```
void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);
```

如果你希望代码在 1.2 版和 2.0 版的 Linux 上都能运行，那最好还是不要直接使用函数*__get_free_pages*，因为它的调用方式在这两个版本间修改过 2 次。只使用函数*get_free_page*(和*__get_free_page*)更安全更可移植，而且也足够了。

至于 DMA，由于 PC 平台设计上的一些“ 特殊性 ”，要正确寻址 ISA 卡还有些问题。我在第 13 章的“ 直接内存访问 ”一节中介绍 DMA 时，将只限于 2.0 版内核上的实现，以避免引入移植方面的问题。

分配函数中的**priority**参数和*kmalloc*函数中含义是一样的。*__get_free_pages*函数中的**dma**参数是零或非零；如果不是零，那么对分配的页面簇可以进行DMA传输。**order**是你请求分配或释放的内存空间相对 2 的幂次(即$\log_2 N$)。例如，如果需要 1 页，**order**为 0；需要 8 页，**order**为 3。如果**order**太大，分配就会失败。如果你释放的内存空间大小和分配得到的大小不同，那么有可能破坏内存映射。在Linux目前的版本中，**order**最大为 5(相当于 32 个页)。总之，**order**越大，分配就越可能失败。

这里值得强调的是，可以使用类似 *kmalloc* 函数中的 **priority** 参数调用 *get_free_pages* 和其他这些函数。某些情况下内存分配会失败，最经常的情形就是优先权为 **GFP_ATOMIC** 的时候。因此，调用这些函数的程序在分配出错时都应提供相应的的处理。

我们已经说过，如果不怕冒险的话，你可以假定按优先权 **GFP_KERNEL** 调用 *kmalloc* 和底层的 *get_free_pages* 函数都不会失败。*一般*说来这是对的，但有时也未必：我那台忠实可靠的 386，有着 4MB 的空闲的 RAM，但当我运行一个"play-it-dangerous"(冒险)模块时却象疯了一样。除非你有足够的内存，想写个程序玩玩，否则我建议你总检查检查调用分配函数的结果。

尽管 **kmalloc(GFP_KERNEL)**在没有空闲内存时有时会失败，但内核总是尽可能满足该内存分配请求。因此，如果分配太多内存，系统的响应性能很容易就会降下来。例如，如果往 *scull* 设备写入大量数据，计算机可能就会死掉；为满足 *kmalloc* 分配请求而换出内存页，系统就会变得很慢。所有资源都被贪婪的设备所吞噬，计算机很快就变的无法使用了；因为此时已经无法为你的 shell 生成新的进程了。我没有在 *scull* 模块中提到这个问题，因为它只是个例子模块，并不能真的在多用户系统中使用。但作为一个编程者，你必须要小心，因为模块是特权代码，会带来系统的安全漏洞(比如说，很可能会造成 DoS("denail-of-service")安全漏洞)。

使用一整页的 *scull*: *scullp*

至此，我们已经较完全地介绍了内存分配的原理，下面我会给出一些使用了页面分配技术的程序代码。*scullp* 是 *scull* 模块的一个变种，它只实现了一个裸(bare)设备－持久性的内存区域。和 *scull* 不同，*scullp* 使用页面分配技术来获取内存；scullp_order 变量缺省为 0，也可以在编译时或装载模块时指定。在 Linux 1.2 上编译的 *scullp* 设备在 order 大于零时会据绝被加载，原因我们前面已经说明过了。在 Linux 1.2 上，*scullp* 模块只允许“安全的”单页的分配函数。

尽管这是个实际的例子，但值的在这提到的只有两行代码，因为该设备其实只是分配和释放函数略加改动的 *scull* 设备。下面给出了分配和释放页面的代码行及其相关的上下文：

```
/*  此处分配一个单位内存  */
  if (!dptr->data[s_pos]) {
      dptr->data[s_pos] = (void *)__get_free_pages(GFP_KERNEL, dptr->order,0);
      if (!dptr->data[s_pos])
              return -ENOMEM;
      memset(dptr->data[s_pos], 0, PAGE_SIZE << dptr->order);

              /*  这段代码释放所有分配单元  */
              for (i = 0; i < qset; i++)
                    if (dptr->data[i])
                          free_pages((unsigned long)(dptr->data[i]), dptr->order);
```

从用户的角度看，可以感觉到的差异就是速度快了一些。我作了写测试，把 4M 字节的数据从 *scull0* 拷贝到 *scull1*,然后再从 *scullp0* 拷贝到 *scullp1*；结果表明内核空间处理器的使用率有所提高。

但性能提高的并不多，因为 *kmalloc* 设计得也运行得很快。基于页的分配策略的优点实际不在速度上，而是更有效地使用了内存。按页分配不会浪费内存空间，而用 *kmalloc* 函数则会浪费一定数量的内存。事实上，你可能会回想起第 5 章的“ 所使用的数据结构 ”一节中我们已经提到过 **select_table** 用了 *__get_free_page* 函数。

使用 *__get_free_page* 函数的最大优点是这些分配得到页面完全属于你，而且在理论上可以通过适当地调整页表将它们合并成一个线性区域。结果就允许用户进程对这些分配得到的不连续内存区域进行 *mmap*。我将在第 13 章的“ mmap 设备驱动程序操作 ”一节中讨论 *mmap* 调用和页表的实现内幕。

vmalloc 和相关函数

下面要介绍的内存分配函数是 *vmalloc*，它分配*虚拟*地址空间的连续区域。尽管这段区域在物理上可能是不连续的(要访问其中的每个页面都必须独立地调用函数 *__get_free_page*)，内核却认为它们在地址上是连续的。分配的内存空间被映射进入内核数据段中，从用户空间是不可见的－这一点上与其他分配技术不同。*vmalloc* 发生错误时返回 0(**NULL** 地址)，成功时返回一个指向一个大小为 **size** 的线性地址空间的指针。

该函数及其相关函数的原型如下：

```
void* vmalloc(unsigned long size);
void vfree(void* addr);
void* vremap(unsigned long offset, unsigned long size);
```

注意在 2.1 版内核中 *vremap* 已经被重命名为 *ioremap*。而且，Linux 2.1 引入了一个新的头文件，<**linux/vmalloc.h**>，使用 *vmalloc* 时应将它包含进来。

与其他内存分配函数不同的是，*vmalloc* 返回很“ 高 ”的地址值－这些地址要高于物理内存的顶部。由于 *vmalloc* 对页表调整后允许用连续的“ 高 ”地址访问分配得到的页面，因此处理器是可以访问返回得到的内存区域的。内核能和其他地址一样地使用 *vmalloc* 返回的地址，但程序中用到的这个地址与地址总线上的地址并不相同。

用 *vmalloc* 分配得到的地址是不能在微处理器之外使用的，因为它们只有在处理器的分页单元之上才有意义。但驱动程序需要真正的物理地址时(象外设用以驱动系统总线的 DMA 地址)，你就不能使用 *vmalloc* 了。正确使用 *vmalloc* 函数的场合是为软件分配一大块连续的用于缓冲的内存区域。注意 *vmalloc* 的开销要比 *__get_free_pages* 大，因为它处理获取内存还要建立页表。因此，不值得用 *vmalloc* 函数只分配一页的内存空间。

使用 *vmalloc* 函数的一个例子函数是 *create_module* 系统调用，它利用 *vmalloc* 函数来获取被创建模块需要的内存空间。而在 *insmod* 调用重定位模块代码后，将会调用

memcpy_fromfs 函数把模块本身拷贝进分配而得的空间内。

用 *vmalloc* 分配得到的内存空间用 *vfree* 函数来释放，这就象要用 *kfree* 函数来释放 *kmalloc* 函数分配得到的内存空间。

和 *vmalloc* 一样，*vremap*(或 *ioremap*)也建立新的页表，但和 *vmalloc* 不同的是，*vremap* 实际上并不分配内存。*vremap* 的返回值是个虚拟地址，可以用来访问指定的物理内存区域；得到的这个虚拟地址最后要调用 *vfree* 来释放掉。

vremap 用于将高内存空间的 PCI 缓冲区映射到用户空间。例如，如果 VGA 设备的帧缓冲区被映射到地址 0xf0000000(典型的一个值)后，*vremap* 就可以建立正确的页表让处理机可以访问。而系统初始化时建立的页表只是用于访问低于物理地址空间的内存区域。系统的初始化过程并不检测 PCI 缓冲区，而是由各个驱动程序自己负责管理自己的缓冲区；PCI 的细节将在第 15 章“ *外设总线概貌* ”的“ PCI 接口 ”一节中讨论。另外，你不必重映射低于 1MB 的 ISA 内存区域，因为这段内存空间可用其他方法访问，参见第 8 章“ *硬件管理* ”的“ 访问设备卡上的内存 ”一节。

如果你希望驱动程序能在不同的平台间移植，那么使用 *vremap* 时就要小心。在一些平台上是不能直接将 PCI 内存区域映射到处理机的地址空间的，例如 Alpha 上就不行。此时你就不能象普通内存区域那样地对重映射区域进行访问，你要用 *readb* 函数或者其他一些 I/O 函数(可参见第 8 章的“ 1M 内存空间之上的 ISA 内存 ”一节)。这套函数可以在不同平台间移植。

对 *vmalloc* 和 *vremap* 函数可分配的内存空间大小并没有什么限制，但为了能检测到程序员的犯下的一些错误，*vmalloc* 不允许分配超过物理内存大小的内存空间。但是记着，*vmalloc* 函数请求过多的内存空间会产生一些和调用 *kmalloc* 函数时相同的问题。

vremap 和 *vmalloc* 函数都是面向页的(它们都会修改页表)；因此分配或释放的内存空间实际上都会上调为最近的一个页边界。而且，*vremap* 函数并不考虑如何重映射不是页边界的物理地址。

vmalloc 函数的一个小缺点是它不能在中断时间内使用，因为它的内部实现调用了 **kmalloc(GFP_KERNEL)**来获取页表的存储空间。但这不是什么问题－如果 *__get_free_page* 函数都还不能满足你的中断处理程序的话，那你还是先修改一下你的软件设计吧。

使用虚拟地址的 *scull*: *scullv*

使用了 *vmalloc* 的示例程序是 *scullv* 模块。正如 *scullp*，这个模块也是 *scull* 的一个变种，只是使用了不同的分配函数来获取设备用以储存数据的内存空间。

该模块每次分配 16 页的内存(在 Alpha 上是 128KB，x86 上是 64KB)。这里内存分配用了较大的数据块，目的是获取比 *scullp* 更好的性能，并且表明此时使用其他可行的分配技术相对来说会更耗时。用 *__get_free_pages* 函数来分配一页以上的内存空间容易出错，而且即使成功了，也相对较慢。前面我们已经看到，用 *vmalloc* 分配若干页比其他函数要快一些，

但由于存在建立页表的开销，只分配一页时却会慢一些。*scullv* 设计得和 *scullp* 很相似。**order** 参数指定分配的内存空间的“幂”，缺省为 4。*scullv* 和 *scullp* 的唯一差别在下面一段代码：

```
/*  此处用虚拟地址来分配一个单位内存  */
 if (!dptr->data[s_pos]) {
     dptr->data[s_pos] = (void *)vmalloc(PAGE_SIZE <<order);
     if (!dptr->data[s_pos])
             return -ENOMEM;

             /*  这段代码释放所有分配单元  */
             for (i = 0; i < qset; i++)
                 if (dptr->data[i])
                     vfree (dptr->data[i]);
```

如果你在编译这两个模块时都打开了调试开关，就可以通过读它们在*/proc* 下创建的文件来查看它们进行的数据分配。下面的快照取自我的计算机，我机器的物理地址是从 0 到 0x1800000(共 24MB)：

```
morgana.root# cp /bin/cp /dev/scullp0
morgana.root# cat /proc/scullpmem
Device 0: qset 500, order 0, sz 19652
   Item at 0063e598, qset at 006eb018
        0: 150e000
        1:   de6000
        2: 10ca000
        3:   e19000
        4:   bd1000

morgana.root# cp /zImage.last /dev/scullv0
morgana.root# cat /proc/scullvmem
Device 0: qset 500, order 4, sz 289840
   Item at 0063ec98, qset at 00b3e810
        0: 2034000
        1: 2045000
        2: 2056000
        3: 2067000
        4: 2078000
```

从这些值可以看到，*scullp* 分配物理地址(小于 0x1800000)，而 *scullv* 分配虚拟地址(但注意实际数值与 Linux 2.1 会不同，因为虚拟地址空间的组织形式变了－见第 17 章“*近期发展*”的“虚拟内存”一节)。

“脏”的处理方法(Playing Dirty)

如果你确实需要大量的连续的内存用作缓冲区，最简单的(也是最不灵活的，但也最容易出错的)方法是在系统启动时分配。显然，模块不能在启动时分配内存；只有直接连到内核的设备驱动程序才能运行这种“脏”的处理方式，在启动时分配内存。

尽管在启动时就进行分配似乎是获得大量内存缓冲区的唯一方法，但我还会在第 13 章的“分配 DMA 缓冲区”一节中介绍到另一种分配技术(虽然可能更不好)。在启动时分配缓冲区有点“脏”，因为它跳过了内核内存管理机制。而且，这种技术普通用户无法使用，因为它要修改内核。绝大多数用户还是愿意装载模块，而并不愿意对内核打补定或重新编译内核。尽管我不推荐你使用这种“分配技术”，但它还是值得在此提及的，因为在 **GFP_DMA** 被引入之前，这种技术曾是 Linux 的早期版本里分配可用于 DAM 传输的缓冲区的唯一方法。

让我们先看看启动是是如何进行分配的。内核启动时，它可以访问系统所有的内存空间。然后以空闲内存区域的边界作为参数，调用内核的各个子系统的初始化函数进行初始化。每个初始化函数都可以“偷取”一部分空闲区域，并返回新的空闲内存下界。由于驱动程序是在系统启动时进行内存分配的，所以可以从空闲 RAM 的线性数组获取连续的内存空间。

除了不能释放得到的缓冲区，这种内存分配技术还有些缺点。驱动程序得到这些内存页后，就无法将它们再放到空闲页面池中了；页面池是在已经物理内存的分配结束后才建立起来的，而且我也不推荐象这样“黑客”内存管理的内部数据结构。但另一方面，这种技术的优势是，它可以获取用于 DMA 传输等用途的一段连续区域。目前这也是分配超过 32 页的连续内存缓冲区的唯一的“安全”的方式，32 页这个值是源于 *get_free_pages* 函数参数 order 可取的最大值为 5。但是如果你需要的多个内存页可以是物理上不连续的，最好还是用 *vmalloc* 函数。

如果你真要在启动时获取内存的话，你必须修改内核代码中的 *init/main.c* 文件。关于 *main.c* 文件的更多细节可参见第 16 章和第 8 章的“1M 内存空间之上的 ISA 内存”一节。

注意，这种“分配”只能是按页面大小的倍数进行，而页面数不必是 2 的某个幂次。

快速参考

与内存分配有关的函数和符号列在下面：

#include <linux/malloc.h>
void *kmalloc(unsigned int size, int priority);
void kfree(void *obj);
这两个函数是最常用的内存分配函数。

#include <linux/mm.h>
GFP_KERNEL
GFP_ATOMIC

GFP_DMA

kmalloc 函数的优先权。**GFP_DMA** 是个标志位，可以与 **GFP_KERNEL** 和/或 **GFP_ATOMIC** 相或。

unsigned long get_free_page(int priority);
unsigned long __get_free_page(int priority);
unsigned long __get_dma_pages(int priority, unsigned long order);
unsigned long __get_free_pages(int priority, unsigned long order,int dma);

这些都是面向页的内存分配函数。以下划线开头的函数不清零分配而得的页。只有前两个函数是在 1.2 版和 2.0 版的 Linux 间可移植的，而后两者在 1.2 版与 2.0 版中的行为并不同。

void free_page(unsigned long addr);
void free_pages(unsigned long addr, unsigned long order);

这些函数用于释放面向页的分配得到的内存空间。

void* vmalloc(unsigned long size);
void* vremap(unsigned long offset, unsigned long size);
void vfree(void* addr);

这些函数分配或释放连续的虚拟地址空间。*vremap* 用*虚拟*地址访问物理内存(在 2.1 版的 Linux 中被称为 *ioremap*)，而 *vmalloc* 是用来分配空闲页面。两种情况下，都是用 *vfree* 来释放分配的内存页。2.1 版的 Linux 引入了头文件<**linux/vmalloc.h**>，使用这些函数时必须先包含(#include)这个头文件。

第 8 章 硬件管理

尽管通过玩玩 *scull* 和类似的一些玩具程序来熟悉 Linux 设备驱动程序的软件接口显得很轻松，但是测试*真正的*设备还是要涉及硬件。驱动程序是软件概念和硬件设备间的一个抽象层；因此，我们两者都要谈谈。到目前为止，我们已经详细讨论了软件上的一些概念；本章将接着介绍驱动程序是如何在保证可移植的前提下访问 I/O 端口和 I/O 地址空间的。

和前面一样，我的示例代码也不针对特定的设备。但是我们不能再用象 *scull* 这样的基于内存的设备。本章的例子使用并口设备来讲解 I/O 指令，使用字符模式 VGA 卡的显示缓冲区来讲解基于内存映射的 I/O。

这里我选择并口的原因是它能提供位信息的输入输出。写向设备的数据位出现在输出引脚上，而输入引脚的电压值可以由处理器控制。实际上，你可以将 LED 连到并口上来观察 I/O 操作的结果。并口比串口更容易编程，而且几乎每台计算机(甚至 Alpha)都和 PC 机一样提供了并口。

至于基于内存映射的 I/O，字符模式的 VGA 是标准的内存映射设备，而每台计算机都有 VGA 兼容的字符模式。遗憾的是，不是每台 Alpha 都有 VGA 显示卡，Sparc 就肯定没有，所以我们的与 VGA 有关的代码就不象并口的例子那样可以移植了。并且，为了运行示例程序，你要切换到字符模式下，但这并不是个太大的限制。用 VGA 显示卡上上内存做实验可能造成的最大问题是示例驱动程序不可避免地会破坏前台的虚拟控制台。

使用 I/O 端口

I/O端口有点类似内存位置：可以用和访问内存芯片相同的电信号对它进行读写。但这两者实际上并不一样；断口操作是直接对外设进行的，和内存相比更不灵活。而且，有 8 位的端口，也有 16 位的端口和 32 位的端口，不能相互混淆[*]。

因此，C 语言程序必须调用不同的函数来访问大小不同的端口。Linux 内核头文件中(就在与体系结构相关的头文件<**asm/io.h**>中)定义了如下一些内联函数。

注意 从现在开始，如果我只使用 unsigned 而不进一步指定类型信息的话，那我是在谈及一个与体系结构相关的定义，此时就不必关心它的准确特性。这些函数基本是可移植的，因为编译器在赋值时会自动进行强制类型转换(cast)－类型被强制转换成 unsigned 类型防止了编译时出现的警告信息。只要程序员赋值时注意避免溢出，这种强制类型转换就不会丢失信息。在本章剩余部分我会一直保持这种“不完整的类型定义”的方式。

---

[*] 实际上，有时I/O端口*是*和内存一样对待的，(例如)你可以将 2 个 8 位的操作合并成一个 16 位的操作。例如，PC的显示卡就可以，但一般来说不能认为一定具有这种特性。

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);

按字节(8 位宽度)读写端口。**port** 参数在一些平台上定义为 **unsigned long**，而在另一些平台上定义为 **unsigned short**。不同平台上 *inb* 返回值的类型也不相同。

unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);

这些函数用于访问 16 位端口(“ 字宽度 ”)；Linux 的 M68k 版本不提供，因为该处理器只支持字节宽度的 I/O 操作，不支持字宽度或更大宽度的操作。

unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);

这些函数用于访问 32 位端口。**doubleword** 参数根据不同平台定义成 **unsigned long** 类型或 **unsigned int** 类型。

除了每次只能传输一个数据单位的 in 和 out 操作，绝大多数处理器还提供了能传输多个字节，字或 long 类型数据的特殊指令。这些指令就是所谓的“ 串指令 ”，将在本章稍后处的“ 串操作 ”一节中介绍。

注意这里没有定义 64 位的 I/O 操作。即使在 64 位的体系结构上，I/O 端口也只使用 32 位的数据通路。

上面这些函数主要是提供给设备驱动程序使用的，但它们也可以在用户空间使用(预处理定义和内联声明没有用**#ifdef __KERNEL__**保护)。但是如果要在用户空间代码中使用 inb 及其相关函数，必须满足下面这些条件：

- 编译该程序时必须带-O 选项来强制内联函数的展开。
- 必须用 *ioperm* 或 *iopl* 来获取对端口进行 I/O 操作的权限。*ioperm* 用来获取对指定端口的操作权限，而 *iopl* 用来获取对整个 I/O 空间的操作权限。这两个函数都是 Intel 平台提供的。
- 必须以 root 身份运行该程序才能调用 *ioperm*。或者，该程序的某个祖先已经以 root 身份获取了对端口操作的权限。

示例程序 *misc-progs/inp.c* 和 *misc-progs/outp.c* 是在用户空间通过命令行读写 8 位端口的一个小工具。我已经在我的 PC 上成功运行过。但由于缺少 *ioperm* 函数的原因，它们不能在其它平台上运行。如果你想冒险，可以将它们设置上 SUID 位，那么不用显式地获取特权就可以使用硬件了。

平台相关性

如果你考虑移植问题，你会发现 I/O 指令是所有计算机指令中与处理器最密切相关的部分。因此，大部分与 I/O 端口有关的源代码都与平台相关。

Linux 系统，尽管是可移植的，但处理器的特性不是完全透明的。大部分硬件驱动程序在平台间是不可移植的，而且在同一模块中驱动程序所涵盖的平台一般不超过两到三种。

回头看看前面的函数列表，你可以看到一处不兼容的地方，数据类型，参数类型根据各平台体系结构上的不同要相应地使用不同的数据类型。例如，port 参数在 x86 平台(处理器只支持 64KB 字节的 I/O 空间)上定义为 **unsigned short**，但在 Alpha 平台上定义为 **unsigned long**。Alpha 平台上端口是和内存在同一地址空间内的一些特定区域，它在设计上就不存在 I/O 地址空间，它的端口是被当作为不能被高速缓冲的内存区域。

I/O 数据类型是核心中一个仍需要整理的部分，尽管现在能正常工作。对这些不够明确的类型最好的解决方法是定义一个与体系结构有关的 **port_t** 数据类型而对数据项则使用 **u8**，**u16** 和 **u32** 这些数据类型(参见第 10 章“ *合理使用数据类型* ”的“ 分配确定的空间大小给数据项 ”一节)。但还没人真正注意过这个问题，因为这个问题主要是个书写技巧上的问题。

其他一些与平台相关的问题来源于处理器结构根本上的差异，因此也无法避免。因为本书我假定你不会在不了解底层硬件的情况下为特定的系统写驱动程序，所以我不会详细讨论这些差异。下面是可以支持的体系结构的总结：

X86
该体系结构支持本章提到的所有函数。

Alpha
支持前面所有函数，但不同的 Alpha 平台上端口 I/O 操作的实现也有不同。串操作是用 C 语言实现的，在文件 *arch/alpha/lib/io.c* 中定义。但遗憾的是，2.0 系列的核心在 2.0.29 版之前还只开放 word 和 long 类型数据的串操作；因此，模块中无法使用 *insb* 和 *outsb* 函数。在 2.0.30 和 2.1.3 版中这个问题已经改正过来了。

Sparc
Sparc 不提供特殊的 I/O 指令。I/O 空间是通过内存映射获得，在页表中设置了标志。在头文件中 inb 和其它函数都被定义为空函数来避免第一次把驱动程序移植到 Sparc 体系结构上时编译器为此报告错误。

M68k
只支持 *inb*，*outb* 和它们相应的暂停式版本(见下节)。68000 上没有定义串操作，也没定义 *readb*，*writeb* 和相关函数。

Mips
支持前面所有函数。但串操作是用汇编语言写的紧凑循环(tight loop)实现的，因为 Mips 处理器不提供机器一级的串 I/O 操作。

PowerPC
除了串 I/O 操作，其它函数都能支持。

感兴趣的读者可以从 *io.h* 文件获得更多信息，除了我在本章介绍的函数，一些与体系结构相关的函数有时也由该文件定义。

值得提及的是，Alpha 处理器并不为端口提供不同的地址空间，虽然 AXP 机器一般带有 ISA 和 PCI 插槽，而且这两种总线都为内存和 I/O 操作提供了不同的信号线。利用特别的接口芯片将指定的内存地址引用转换成对 I/O 端口的访问，基于 Alpha 的 PC 可以实现一个与 Intel 系列兼容的 I/O 抽象层。

Alpha 平台上的 I/O 操作在“ Alpha 参考手册 ”中详细介绍了，该手册可从 DEC 公司免费获得，手册详尽地阐述了 I/O 问题，介绍了 AXP 处理器是如何将虚拟地址空间划分为“ 类内存 ”和“ 不类内存 ”区域；后者用于内存映射的 I/O。

暂停式(pausing)I/O

一些平台－特别是在 i386 上－当处理器和总线间数据得传输太快是会带来问题。问题是源于相对 ISA 总线处理器的时钟频率太快了，当设备卡太慢时，这个问题就容易暴露出来；解决该问题的方法是，如果后面又跟着一条 I/O 指令，就在该条 I/O 指令后添加一小段延迟。如果你的设备会丢失数据，或者你担心它会丢失数据，你可以用暂停式的 I/O 操作取代通常的 I/O 操作。暂停式 I/O 函数很象前面列出的那些 I/O 函数，但它们的名字都以 *_p* 结尾；例如 *inb_p*，*outb_p* 等等。对所有支持的体系结构，如果定义了不暂停的 I/O 函数，那么也会定义相应的暂停式的 I/O 函数，虽然有些平台上它们会被扩展成相同的代码。

如果你想在驱动程序中显式地插入一小段延迟(小于用 *udelay* 可获得的延迟)，你可以显式地使用 **SLOW_DOWN_IO** 语句。由这个宏扩展成的一段指令只是用于延迟，不做任何其他的工作。你可以将这条语句插入到源代码中的一些关键位置上。实际上，**SLOW_DOWN_IO** 所执行的代码就是 *outb_p* 被扩展后相对于 *outb* 所增加的代码。

SLOW_DOWN_IO(和 **_p** 延迟)是否被定义取决于在包含<**asm/io.h**>文件前是否定义过 **SLOW_IO_BY_JUMPING** 和/或 **REALLLY_SLOW_IO**。好在新的硬件已经不要求程序处理这些问题了，所以我不会再讨论暂停问题了。感兴趣的读者可以读读头文件<**asm/io.h**>。但是在写驱动程序的时候，你必须记着 **SLOW_DOWN_IO** 在 Sparc 和 M68k 体系结构上是没有定义的(虽然也定义了象 *outb_p* 这样暂停式的函数调用，在本章前面的“ 平台相关性 ”一节中列出了限制条件)。

串操作

Linux 头文件中定义了进行串操作的函数，驱动程序可以使用它们来获得比 C 语言写的循环更好的性能。在 Linux 中，取决与相应的处理器或平台，串操作被实现为一条机器指令，或者是一个紧凑循环，但也可能不提供。

串操作函数的原型如下：

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
从内存地址 **addr** 开始连续读写 **count** 个字节。但只对一个端口 **port** 读写这些数据。

void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
对一个 16 位端口连续读写 16 位数据。

void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);
对一个 32 位端口连续读写 32 位数据。

使用并口

我们用并口来测试我们的 I/O 代码，并口很简单，事实上，我很难想出一个比它更简单的接口适配卡。

尽管大部分读者都能取得并口规范，但为了方便读者，在你读下面要介绍的模块之前我先将它总结一下。

并口的基本知识

并口的最小配置(不涉及 ECP 和 EPP 模式)由一些 8 位端口组成。写到输出端口的数据表现为 25 脚插座的输出引脚上的电平信号，而从输入端口读到的数据则是输入引脚当前的逻辑电平值。

在并行通信中使用的电平信号是标准的 TTL 电平：0 伏和 5 伏，逻辑阈值大约为 1.2 伏；端口要求至少满足标准的 TTL LS 电流规格，而现代的大部分并口电流和电压都超过这个规格。

警告 并口插座没有和计算机的内部电路分开，这一点在你想把逻辑门直接连到端口时很有用。但要注意正确连线；否则在测试自己定制的电路时，并口很容易被烧毁。如果你担心会破坏你的主板的话，可以选用可插拔的并行接口。

位规范显示在图 8-1 中。你可以读写 12 个输出位和 5 个输入位，其中一些位在它们的信号通路上输入输出会有逻辑上的反转。唯一一个不与任何信号引脚有联系的位是 2 号端口的第 4 位(0x10)。我们将在第 9 章“ *中断处理* ”中使用到它。

控制端口：基地址+2
状态端口：基地址+1
数据端口：基地址+0

关键字
输入信号线
输出信号线
位
引脚
反转 非反转
图 8-1：并口的插脚引线

驱动程序样例

我下面要介绍的驱动程序叫做 *short*(Simple Hardware Operations and Raw Tests，简单硬件的操作和原始测试)。它所做的就是读写并口(或其他 I/O 设备)的各种 8 位端口。每个设备节点(拥有唯一的次设备号)访问一个不同的端口。*short* 设备没有任何实际用途；将它独立出来只是为了能用一条指令来从外部对端口进行操作。如果你不太了解 I/O 端口，那么可以通过使用 *short* 来熟悉它；你可以测量它传输数据要消耗的时间或者进行其它的测试。

我建议你把一个 LED 焊到输出引脚上以便观察并口插座。每个 LED 都要串联一个 1KΩ的电阻到一个接地的引脚上。如果将输出端口接到输入端口上，你就可以产生自己的输入供输入端口读取。

如果你想将 LED 焊到 D 型插座上来观察并行数据，我建议你不要使用 9 号和 10 号引脚，因为在运行第 9 章的示例代码时我们要将它们相连。

至于 *short*，它是通过紧凑循环将用户数据拷贝到输出端口来实现写设备*/dev/short0* 的，每次一个字节：

```
while (count--)
    outb(*(ptr++), port);
```

你可以运行下面的命令来使你的 LED 发光：

```
echo -n any string > /dev/short0
```

每个 LED 监控输出端口的一个位。注意只有最后写的字符数据才会在输出引脚上稳定地保持下来，被你观察到。因此，我建议你将*-n* 选项传给 *echo* 程序来制止输出字符后的自动换行。

读端口也是使用类似的函数，只是用 *inb* 代替了 *outb*。为了从并口读取“ 有意义的 ”值，你需要将某个硬件连到并口插座的输入引脚上来产生信号。如果没有输入信号，你只会读到始终是相同字节的无穷的输出流。

覆盖所有可能的 I/O 函数，每个 *short* 设备都提供了三个变种：*/dev/short0* 执行的是上面给出的循环，*/dev/short0p* 使用 *outb_p* 和 *inb_p* 来替代前者使用的“ 较快的 ”函数，而 */dev/short0s* 使用了串指令。共有四个这样的设备，从 *short0* 到 *short3*，每个都读写一个端口。

这四个端口是连续的。

在 Alpha 上编译时，由于不提供 *insb* 和 *outb*，*short0s* 设备就和 *short0* 一样了。

虽然 *short* 不能进行“真正的”硬件控制，但它可以作为一个对不同的指令计时的有趣的测试平台，并且可以作为一个学习如何进行“真正的”硬件控制的开始。任何对写驱动程序有兴趣的人肯定拥有更多更有趣的设备，但老的傻瓜式的并口还是能作些有用的工作的－我自己就是在早上打开收音机后用它来为我准备咖啡的。

访问设备卡上的内存

上一章介绍了分配 RAM 内存的所有各种方法；现在我们要介绍计算机能提供的另一种内存：扩展卡上的内存。外设也有内存。显卡有帧缓冲区，视频捕捉卡用来存放捕捉到的数据，而网卡将接受到的数据包存放在内存区域中；此外，大多数设备卡上还有卡上 ROM，存放着系统启动时处理器要执行的代码。所有这些都是“内存”，因为处理器通过访存指令来对它们进行读写。这里我只讨论 ISA 和 PCI 设备，因为现在它们最常用。

在标准的 x86 机器上共有三种通用的设备内存：640KB 到 1MB 地址范围内的 ISA 内存，14MB 到 16MB 地址范围内的 ISA 内存，和在物理地址空间之上的 PCI 内存。上面这些用到的地址都是物理地址，是在计算机的地址总线上的跑的数值，与程序代码中所使用的虚拟地址没有任何关系(参见第 7 章“*获取内存*”的“vmalloc 和相关函数”一节)。I/O 内存使用这些物理位置主要是出于历史的原因，下面介绍这三种内存区域时会作出解释。

不幸的是(或者，幸运的是，如果你更倾向于好的体系结构设计而不是容易移植的话)。不是所有的 Linux 平台都支持 ISA 和 PCI；本节只限于讨论支持 ISA 和 PCI 的 Linux 平台。

1M 地址空间之下的 ISA 内存

在第 2 章“*编写和运行模块*”的“ISA 内存”一节中我已经介绍过一种“容易的”(但有些毛病的)访问这种内存的方法，在那里我使用了存放物理地址的指针来正确地指向所申请的 I/O 内存。尽管这种技术在 x86 平台可行，但却不能移植到其它的 Linux 平台上。使用指针的方法对小的生命期较短的项目是较好的选择，但对作为产品的驱动程序并不推荐。

推荐的 I/O 内存接口是在 linux1.3 版的开发树中被引入内核的，更早的版本并不提供。与 *short* 范例模块一起发布的 *sysdep.h* 头文件为 1.2 版以来的各个内核版本实现了这种新的语义。

新的接口包括一系列宏和函数调用，取代了使用指针来访问 I/O 内存的方法。这些宏和函数是可移植的；这意味着相同的源码可以在不同的体系结构上编译和运行，只要它们拥有类型相同的设备总线。

当你在基于 Intel 处理器的平台上编译代码时，这些宏现在大部分会扩展对指针的操作，但它们的内部实现会在将来被适当地修改。例如，在 Linux 最初从 2.0 版转到 2.1 版时，这

样的修改就发生过，Linus 决定改变虚存的布局。在新的布局下，就不能再以第 2 章中介绍的旧的方式访问 ISA 内存了。

新的 I/O 内存接口由下面这些函数组成：

unsigned readb(address);
unsigned readw(address);
unsigned readl(address);

这些宏用于从 I/O 内存中取得 8 位，16 位和 32 位的数据值。1.2 版的 Linux 不提供。使用宏的优点是对参数的类型不作要求；参数 **address** 在使用前会被强制类型转换，因为这个值“不清楚是整数还是指针，但我们两者都能接受”(见 *asm-alpha/io.h*)。读和写函数都不会检查 address 参数的合法性，因为使用它就是为了能和使用指针一样快(我们已经知道有时它们实际上就是被扩展成指针操作)。

unsigned writeb(unsigned value, address);
unsigned writew(unsigned value, address);
unsigned writel(unsigned value, address);

和前面的函数类似的，这些函数(宏)用于写 8 位，16 位和 32 位的数据项。

memset_io(address,value,count);

当你要对 I/O 调用 *memset* 进行操作时，这个函数可以满足你的需要，并且也保留了 *memset* 原来的语义。

memcpy_fromio(dest, source, nbytes);
memcpy_toio(dest, source, nbytes);

这些函数用于成块传输 I/O 内存的数据，和 *memcpy_tofs* 的功能有些相似。它们是和上面这些函数一起引入 Linux 中的，1.2 版的 Linux 不提供。与示例代码一起发布的 *sysdep.h* 头文件修正了函数的版本相关性问题，为 1.2 版以后的所有内核提供了这些函数的定义。

和 I/O 端口函数一样的，这些函数在能支持的体系结构间的移植性现在也很有限。一些平台根本不提供这些函数；一些平台上它们是被扩展为指针操作的宏，而在另一些平台上它们则是真正的函数。

象我这种习惯于旧 PC 机的平的(flat)内存模式的人，可能会不愿意为了只是读写“物理地址区域”而麻烦地使用新的一套接口。实际上，要习惯使用某套接口只需要练习练习使用这些函数。当然，没有比看看一个访问 I/O 内存的傻瓜式(silly)模块更好的获得自信的方法了。我下面要给你展示的模块就叫作 *silly*，是“Simple Tool for Unloading and Printing ISA Data，卸载和打印 ISA 数据的简单工具”的简称。

该模块包括四个用了不同的数据传输函数来完成相同任务的设备节点。*silly* 设备是作为 I/O 内存之上的一个窗口，与 */dev/mem* 有些类似。对该设备，你可以读写数据，*lseek* 到一个任意的 I/O 内存地址，也可以将 I/O 内存区域 *mmap* 到你的进程中来(参见第 13 章“*Mmap 和 DMA*”的“mmap 设备操作”一节)。

/dev/sillyb，次设备号为 0，调用 *readb* 和 *writeb* 函数来读写内存空间。下面的代码给出了 *read* 的实现，它将 0xA0000 到 0xFFFFF 的地址范围重映射到设备文件的偏移量从 0 到 0x5FFFF 的位置。*read* 函数将不同的访问模式组织进一个 **switch** 语句中；下面是 *sillyb* 设备的 **case** 分支：

```
case M_8:
    while (count){
        *ptr=readb(add);
        add++;count--;ptr++;
    }
    break;
```

接下来的两种设备是*/dev/sillyw*(次设备号为 1)和*/dev/sillyl*(次设备号为 2)。它们和 */dev/sillyb* 设备差不多，只是分别使用了 16 位和 32 位的函数。下面是 *sillyl* 设备的 *write* 函数的实现，也是 **switch** 语句的一部分。

```
case M_32:
    while (count>=4 ){
        writel(*(u32*)ptr,add);
        add+=4;count-=4;ptr+=4;
    }
    break;
```

最后一种设备是*/dev/sillycp*(次设备号为 3)，该设备用 *memcpy_*io* 函数来执行相同的任务。下面是它的 *read* 函数实现的核心：

```
case M_memcpy:
    memcpy_fromio(ptr, add, count);
    break;
```

1M 地址空间之上的 ISA 内存

有些 ISA 设备卡带有的卡上内存会映射到物理地址空间的 14MB 和 16MB 范围内。这些设备正在逐渐消失，但仍值得介绍一下如何访问它们的内存区域。但本讨论仅适用于 x86 体系结构；我没有这种 ISA 卡在 Alpha 或其它体系结构上相应行为方面的资料。

在使用 80286 处理器的旧时代，物理地址空间的宽度是 20 个位(16MB)，所有的地址线都在 ISA 总线上，几乎没有计算机带的 RAM 会超过 1 兆或 2 兆。那为什么扩展卡不能“ 偷 ”点高端的内存地址用做它的缓冲区呢？这种想法并不新鲜 ;相同的概念已经出现在对 1M 地址空间之下的 ISA 内存的使用上了，后面又会再次用来实现对高端 PCI 内存的使用上。选作 ISA 设备卡内存的地址范围是最顶端的 2M，尽管大多数卡只使用了最顶端的 1M。

只到今天，在一些主板仍能使用旧式的设备卡，即使物理内存超过了 14M。要正确地处理这段内存区域要求你对这段地址范围作些处理，避免将内存地址和总线地址重叠起来。

如果你有一块带有高端内存的 ISA 设备，又不幸地 RAM 小于 16MB，那管理内存就很容易。你的软件在处理时就好象拥有着高端 PCI 缓冲区(参见下一节)，只是更慢些，因为 ISA 内存比较慢。

而如果你拥有的 ISA 设备带了高端内存而你的 RAM 要多于 16MB，那么就有麻烦了。

一种可能就是你的主板不能正确地支持“ ISA 空洞 ”。在这种情况下，除非你拔掉一些内存，否则无法访问卡上内存。另一种可能是，主板能处理 ISA 空洞，你仍要告诉 Linux 内核这种内存的存在，作一些处理以便能访问 RAM 的其它部分(超过 16M 的地址范围)。

你需要作些“ 黑客 ”的工作来正确保留高端的 ISA 内存，同时又仍能对其余的 RAM 进行正常访问，要修改的部分是对计算机的物理内存进行映射的部分。这个映射部分是在源文件 *arch/i386/mm/init.c* 的函数 *mem_init* 中实现的。数组 **mem_map** 中存放着与内存页有关的信息；如果某页的 **PG_reserved** 位被设置了，内核就不会对该页进行正常的页面处理(也即，该页被“ 保留 ”了，不要碰它)。但驱动程序仍可以使用保留页；640KB 到 1MB 间的地址范围被标记为“ 保留的 ”，但仍可以被用作为设备内存。

下面的代码，插在 *mem_init* 函数中，正确地保留了 15MB 和 16MB 间的地址空间：

```
while(start_mem<high_memory){
    if (start_mem>=0xf00000 && start_mem<0x1000000){
        /* keep it reserved, and prevent couting as data */
        reservedpages++; datapages--;
    }
    else
        clear_bit(PG_reserved, &mem_map[MAP_NR(start_mem)].falgs);
    start_mem += PAGE_SIZE;
}
```

最初所有的内存区域都被标记为“ 保留的 ”，上面给出的代码行保证了不会将对高端的 I/O 内存去除“ 保留 ”标记；原来的代码只有上面给出的循环的 **else** 分支。因为在内核代码之后的每个保留页都算作内核数据，要修改两个计数器 reservedpages 和 datapages 以避免启动时给出不匹配的信息。我的机器，有 32MB 内存，用前面的代码来访问 ISA 空洞，启动是的报告如下：

```
Memory: 30120/32768k available (512k kernel code, 1408k reserved, 728k data)
```

我是在自己的安装了 2.0.29 版内核的 Intel 机器(主板支持 ISA 空洞)上测试这段代码的。如果你运行的内核版本并不相同，你可能需要修改一下代码－与内存管理有关的内部数据结构在 2.1 版的内核中有一点改动，而在 1.2 版的内核中则很不一样。要支持旧式的(有时是不良设计的)硬件设备不可避免地必须“ 黑客 ”内核代码。

高端 PCI 内存

访问高端 PCI 内存比访问高端 ISA 内存要更容易。PCI 卡上的高端内存*真的*很高－高于任何合理的物理 RAM 地址(至少对未来若干年而言)。

正如第 7 章的“ vmalloc 和相关函数 ”一节讨论到的，访问该内存只要调用 *vremap*(2.1 版的内核是 *ioremap*)。但是如果你希望代码能在不同平台上移植的话，你应该只通过 *readb* 和其它类似函数来访问重映射的内存区域。由于不是所有平台都能将 PCI 缓冲区直接映射到处理器的地址空间，所以必须加上这个限制。

访问字符模式的视频缓冲区

silly 模块解释了如何访问内存 640KB 到 1MB 地址范围内的视频缓冲区，而下面要介绍的更“ 直观的 ”演示程序则能帮助你熟悉 *readb* 和 *writeb* 函数。*silly* 模块拥有两个额外的设备节点：*/dev/sillytxt*(次设备号为 4)和*/dev/sillitest*(次设备号为 5)。

警告 这样的设备只能在运行在字符模式下的 VGA 兼容的显示卡上使用；在没有 VGA 显示卡的系统上使用这样的设备，和任何对硬件资源的不受控制的读写一样，具有潜在的破坏性。

第一个设备，*sillytxt*，只是 VGA 字符缓冲区上的一个窗口。与其它的 *silly* 节点不同的是，它可以作为输出重定向的目标和用于覆盖控制台上的显示。这让我们想起*/dev/vcs*，但 *silly* 的实现是不可移植的而且也没有象 *vcs* 那样集成进内核。

最后一个设备只是和你开个玩笑：它将字母从字符屏幕上移走。每向该设备写进一个字符都会导致屏幕上的一个字符落到屏幕的底部。提供这个设备只是为了演示在 I/O 内存如何进行更复杂的操作－可以用同样的代码来操作 VGA 缓冲区或其它内存，比如网络数据包或者帧捕捉卡上的视频数据。

要注意对字符屏幕的任何修改都是易丧失的，并且会干扰内核自己的字符处理。如果你真的需要在应用程序中访问字符缓冲区，那么有更好的完成这个任务的方法：通过 *ncurses* 库或者通过*/dev/vcs* 设备。*vcs* 设备是“ Virtual Console Screen，虚拟控制台屏幕 ”，可以用它来获得每个虚拟控制台的字符缓冲区当前的快照或者进行修改。*vcs* 设备的文档就在它自己的源码中：内核源码树中的 *drivers/char/vc_screen.c* 文件。或者，你可以在最新的 *man-pages* 帮助页发布中找到关于该设备的描述。

快速参考

本章引入下面一些与操纵硬件有关的符号：

#include <asm/io.h>

unsigned inb(unsigned port);
void outb(unsigned char byte, unsigned port);
unsigned inw(unsigned port);
void outw(unsigned short word, unsigned port);
unsigned inl(unsigned port);
void outl(unsigned doubleword, unsigned port);

这些函数用于读写 I/O 端口。如果能正确获取访问端口的权限，它们也可以被用户空间的程序调用。不是所有平台都支持所有这些函数，它们与底层的硬件设计有关。

SLOW_DOWN_IO；
unsigned inb_p(unsigned port);
…

有时需要用 **SLOW_DOWN_IO** 语句来处理 x86 平台上低速的 ISA 卡。如果在每个 I/O 操作之后需要一小段延迟，你可以用与上面引入的 6 个函数相应的暂停式版本，它们得在相应的函数名后要加个**_p**。

void insb(unsigned port, void *addr, unsigned long count);
void outsb(unsigned port, void *addr, unsigned long count);
void insw(unsigned port, void *addr, unsigned long count);
void outsw(unsigned port, void *addr, unsigned long count);
void insl(unsigned port, void *addr, unsigned long count);
void outsl(unsigned port, void *addr, unsigned long count);

“串操作”用于优化从输入端口到内存区域(或者反过来)的数据传输。这种传输是通过对同一个端口读写 **count** 次完成的。

unsigned readb(address);
unsigned readw(address);
void writeb(unsigned value, address);
void writew(unsigned value, address);
void writel(unsigned value, address);
memset_io(address, value, count);
memcpy_fromio(dest, source, nbytes);
memcpy(dest, source, nbytes);

所有这些函数都用于访问 I/O 内存区域－低端的 ISA 内存或者高端的 PCI 缓冲区(在调用 *vremap* 后)。

第 9 章 中断处理

中断是硬件管理的最终资源。众所周知，设备利用中断来通知软件可以对它进行操作了。Linux 为中断处理提供了很好的接口。事实上中断处理的接口如此之好，以至于编写和安装中断处理程序几乎和编写其它核心函数一样容易。但是由于中断处理程序和系统的其它部分是异步运行的，使用时要注意一些事项。

本章的示例代码使用并口来产生中断。因此，如果你想运行测试程序，你必须给你的电烙铁接上电源，即使在上一章的例子程序中你拒绝这样做。

我们用上一章的 short 模块来示范如何使用中断。这个模块的名字，*short*，实际上是指 *short int*(这是 C 语言，不是吗？)，提醒我们它要对中断(*int*errupt)进行处理。

准备并口

虽然我在第 8 章“ *硬件管理* ”的“ 使用并口 ”一节已经提到，并口很简单，但它也会触发中断。打印机就是利用这种能力来通知 *lp* 驱动程序它已准备好接收缓冲区中的下一个字符。

在指定接口这样做之前实际上并不会产生中断；并口标准规定设置 2 号端口的第 4 位(0x37a，0x27a 或其它某个地址)时启动中断报告。*short* 模块在初始化时调用 *outb* 设置该位。

启动中断报告之后，每当引脚 10(所谓的"ACK"位)上的电平从低变高时，并口都会产生一个中断。强迫接口（没有把打印机连到端口上）产生中断的最简单方法是将并行插座的引脚 9 和引脚 10 相连。为此，你需要一个阳性的 25 针 D 型插座和一英寸的电缆线。

引脚 9 是并行数据字节中最重要的一位。如果你往*/dev/short0* 中写入二进制数据，就可以产生几个中断。然而，往端口中写入 ASCII 文本将不会产生中断，因为此时不会设置这个最重要的位。

如果你确实想“ 看看 ”产生的中断，那么，仅仅往硬件设备中写是不够的；还必须在系统中配置一个软件处理程序。目前，Linux-x86 和 Linux-Alpha 只是简单的确认，忽略任何在预料之外的中断。

安装中断处理程序

中断信号线是宝贵并且非常有限的资源，当只有 15 或 16 根中断信号线时尤其如此。内核维护了一个类似于 I/O 端口注册表的中断信号线的注册表。一个模块可以申请一个中断通道（或中断请求 IRQ，即 Interrupt ReQuest），并且，处理完以后还可以释放掉它。在 **<linux/sched.h>**头文件中申明的下列函数实现了这个接口：

```
int request_irq(unsigned int irq,
                void (*handler)(int, void*, struct pt_regs *),
                unsigned long flags,
                const char *device, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

注意，1.2 版定义了不同的原型。相关的移植问题可参见本章稍后的“ IRQ 处理程序的版本相关性 ”一节。

通常，申请中断的函数的返回值为 0 时表示成功，或者返回一个负的错误码。函数返回 **-EBUSY** 通知另一个设备驱动程序已经使用了要申请的中断信号线的情况并不常见。函数参数定义如下：

unsigned int irq

该参数为中断号。有时从 Linux 中断号到硬件中断号的映射并不是一对一的。例如，在 *arch/alpha/kernel/irq.c* 文件中可以查看到 Alpha 上的映射。这里，传递给内核函数的参数是 Linux 中断号而不是硬件中断号。

void (*handler)(int,void *,struct pt_regs *)

指向要安装的中断处理函数的指针。

unsigned long flags

如你所想，这是一个与中断管理有关的各种选项的字节掩码。

const char *device

传递给 *request_irq* 的字符串，在*/proc/interrupts* 中用于显示中断的拥有者（参见下一节）。

void *dev_id

这个指针用于共享的中断信号线。它是一个唯一的标志符，更象一个 ClientDate（C++ 中的 this 对象）。设备驱动程序可以自由地任意使用 **dev_id**。除非强制使用中断共享，**dev_id** 通常被置为 **NULL**。在后面的“ 实现一个处理程序 ”一节中，我们将看到一个使用 dev_id 的实际例子。

在 **flags** 中可以设置的位是：

SA_INTERRUPT

如果设置该位，就指示这是一个“ 快速 ”中断处理程序；如果清除这位，那么它就是一个“ 慢速 ”中断处理程序。快速中断处理程序和慢速中断处理程序的概念在下面的“ 快速和慢速处理程序 ”一节中会谈到。

SA_SHIRQ

该位表明中断可以在设备间共享。共享的概念在稍后的“ 中断共享 ”一节中介绍。

SA_SAMPLE_RANDOM

该位表明产生的中断对*/dev/random* 和*/dev/urandom* 设备要使用的熵池(entropy pool)有贡献。读这些设备返回真正的随机数，它们用来帮助应用软件选取用于加密的安全钥匙。这些随机数是从一个熵池中取得的，各种随机事件都会对系统的熵池(无序度)有贡献。如果你希望设备真正随机地产生中断，你应该置上这个标志。而如果你的中断是可预测的(例如，帧捕捉卡的垂直消隐)，那就不值得设置这个标志位－它对系统的熵池没有任何贡献。更详尽的信息可参见 *drivers/char/random.c* 文件中的注释。

中断处理程序可以在驱动程序初始化时或者在设备第一次打开时安装。虽然在 *init_module* 函数中安装中断处理程序听起来是个好主意，但实际上并非如此。因为中断信号线数量有限，你不会想浪费它们的。你的计算机拥有的设备通常要比中断信号线多。如果一个设备模块在初始化就申请了一个中断，会阻碍其它驱动程序使用这个中断，即便这个设备根本不使用它占用的这个中断。而在打开设备时申请中断，则允许资源有限的共享。

例如，只要你不同时使用帧捕捉卡和调制解调器这两个设备，它们使用同一个中断就是可能的。用户经常在系统启动时装载某个特殊设备的模块，即使这个设备很少使用。数据采集卡可以和第二个串口使用同一个中断。尽管在进行数据采集时避免去连你的 ISP 并不是件难事，但在使用调制解调器前不得不先卸载一个模块太令人不愉快了。

调用 *request_irq* 的正确位置是在设备第一次打开，硬件被指示产生中断*前*的时候。而调用 *free_irq* 的位置是设备最后关闭，硬件被通知不要再中断处理器*后*的时候。该技术的缺点是你必须为每个设备维护一个记录其打开次数的计数器。而如果你在同一个模块中控制两个以上的设备，那么仅仅使用模块计数器那还不够。

尽管我已说了这么多，*short* 却是在装载时申请中断信号线的。我这样做是为了使你在运行测试程序时不必运行其它进程来使设备保持打开的状态。因此，*short* 会象真正的设备那样，在 *init_module* 中而不是 *short_open* 中申请中断。

下面这段代码要申请的中断是 **short_irq**。对这个变量的赋值将在后面再给出，因为它与现在的讨论无关。**short_base** 是使用的并口的 I/O 基地址；写接口的 2 号寄存器打开中断报告。

```
if (short_irq >=0 ) {
        result=request_irq(short_irq, short_interrupt, SA_INTERRUPT, "short", NULL);
        if (result) {
                printk(KERN_INFO "short: can't get assigned irq %i\n", short_irq);
                short_irq=-1;
        }
        else { /*
                outb(0x10, short_base+2);
        }
}
```

这段代码显示安装的处理程序是个快速中断处理程序(**SA_INTERRUPT**)，不支持中断

共享(没有设置 **SA_SHIRQ**)，并且对系统熵池无贡献(没有设置 **SA_SAMPLE_RANDOM**)。然后调用 *outb* 打开并口的中断报告。

/proc 接口

当处理器被硬件中断时，一个内部计数器会被加 1，这为检查设备是否正常工作提供了一个方法。报告的中断显示在文件*/proc/interrupts* 中。下面是我的 486 启动一个半小时(uptime)后该文件的一个快照：

```
 0:     537598     timer
 1:      23070     keyboard
 2:          0     cascade
 3:       7930 +   serial
 5:       4568     NE2000
 7:      15920 + short
13:          0     math error
14:      48163 + ide0
15:       1278 + ide1
```

第一列是 IRQ 中断号。你可以从显示中缺少一些中断推知该文件只会显示已经安装了驱动程序的那些中断。例如，第一个串口(使用中断号 4)没有显示，这表明我现在没有使用调制解调器。实际上，即使我在获取这个快照之前使用过调制解调器，它也不会出现在这个文件中；串口的行为很良好，当设备关闭时会释放它们的中断处理程序。出现在各记录中的加号标志该行中断采用了快速中断处理程序。

/proc 树中还包含了另一个与中断有关的文件，*/proc/stat*；有时你可能会发现一个文件更有用，但有时又更愿意使用另一个。*/proc/stat* 文件记录了关于系统活动的一些底层的统计信息，包括(但不仅限于)只系统启动以来接收到的中断次数。*stat* 文件的每一行都以一个字符串开始，它是该行的关键字；**intr** 标记正是我们要找的。下面的快照是在得到前面那个快照后半分钟获得的：

```
intr 947102 540971 23346 0 8795 4907 4568 0 15920 0 0 0 0 0 0 48317 1278
```

第一个数是总的中断次数，而其它每个数都代表一个中断信号，从 0 号中断开始。上面的快照显示 4 号中断被使用了 4907 次，虽然*当前*它的处理程序没有安装上。如果你测试的驱动程序是在每次打开和关闭设备的循环中获取和释放中断的话，那么你会发现*/proc/stat* 文件要比*/proc/interrupts* 文件更有用。

两个文件另一处不同是 *interrupts* 文件与体系结构无关，而 *stat* 文件则与体系结构有关：其字段的个数取决于内核之下的硬件。可以获取的中断个数在 Sparc 上只有 15 个，而在 Atari(M68k 处理器)上则多达 72 个。

下面的快照给出我的 Alpha 工作站(共有 16 个中断，和 x86 机器一样)上的文件内容：

```
 1:          2    keyboard
 5:       4641    NE2000
15:      22909 + 53c7,8xx

intr    27555 0 2 0 1 1 4642 0 0 0 0 0 0 0 0 22909
```

这个输出的最值得注意的地方是不出现时钟中断。在 Alpha 机器上，时钟中断与其它中断到达处理器的方式不同，没有分配 IRQ 中断号。

自动检测中断号

驱动程序初始化时最迫切的问题之一就是如何决定设备要使用哪条中断信号线。驱动程序需要该信息以便安装正确的处理程序。虽然程序员可以要求用户在装载是指定中断号，但这并不好，因为一般用户并不知道中断号，或者是因为他没有配置跳线或者因为该设备根本就没有跳线。自动检测中断号是对驱动程序使用的基本要求。

有时自动检测依赖于一些设备拥有的较少改变的缺省特性。此时，驱动程序可以就假定设备使用了这些缺省值。*short* 在检测并口时就正是这么作的。正如 *short* 的代码中所给出的，实现起来相当简明：

```
if (short_irq<0) /*  尚未指定：强制为缺省的  */
      switch(short_base){
            case 0x378: short_irq=7; break;
            case 0x278: short_irq=2; break;
            case 0x3bc: short_irq=5; break;
}
```

这段代码根据选定的 I/O 地址来分配中断号，但也允许用户在装载驱动程序时通过调用 **insmod short short_irq=x** 来覆盖缺省值。**short_base** 缺省为 0x378，因此 **short_irq** 缺省为 7。

有些设备设计得更为先进，会简单地“声明”它们要使用那个中断。此时，驱动程序可以通过读设备的某个 I/O 端口的一个状态字节来获得中断号。当目标设备能告述设备要使用哪个中断时，那么自动检测中断号就是探测设备，不需要额外工作来探测中断。

值得注意的是，现代的设备能提供自己的中断配置信息。PCI 标准通过要求外围设备声明要使用的中断信号线的方法来解决这个问题。关于 PCI 标准的讨论可参见第 15 章“*外设总线概貌*”。

遗憾的是，不是所有设备都对程序员友好，自动检测可能还是需要一些探测的。技术很简单：驱动程序告诉设备产生中断，然后观察会发生些什么。如果一切正常，那么只有一条中断信号线被激活了。

尽管探测在理论上很简单，实际的实现则并不那么简明。下面我们看看执行该任务的两种方法：调用内核定义的帮助函数和实现我们自己的版本。

核心帮助下的检测

主流的内核版本都提供探测中断号的底层工具。这种工具包括两个函数，都在头文件<**linux/interrupt.h**>中声明(该头文件也描述了探测的机制)：

unsigned long probe_irq_on(void);

这个函数返回尚未分配的中断的位掩码。驱动程序必须保留返回的位掩码以便随后能将它传递给 *probe_irq_off* 函数。调用该函数后，驱动程序要安排相应设备至少产生一次中断。

int probe_irq_off(unsigned long);

在设备已经申请了中断之后，驱动程序要调用这个函数，传递给它的参数是先前调用 *probe_irq_on* 返回的位掩码。*probe_irq_off* 返回“启动探测”后发出的中断次数。如果没有发生任何中断，就返回 0(因此无法探测 0 号中断，但在能支持的所有体系结构上也没有什么定制设备能使用它)。如果产生了多次中断(二义性检测)，*probe_irq_off* 将返回一个负值。

程序员要注意在调用 *probe_irq_on* 后启动设备，并在调用 *probe_irq_off* 后关闭它。此外，在调用 *probe_irq_off* 之后，不要忘了处理你的设备尚未处理的那些中断。

short 模块演示了如何进行这样的探测。如果你在装载模块时指定 **probe=1** 并且并口插座的 9 号和 10 号引脚相连，就会执行下面的代码进行中断信号线的检测。

```
int count=0;
do {
    unsigned long mask;

    mask=probe_irq_on();
    outb_p(0x10, short_base+2); /*  启动中断报告  */
    outb_p(0x00,short_base); /*  清位  */
    outb_p(0xFF, short_base);   /*  置位：中断！*/
    outb_p(0x00, short_base+2); /*  关闭中断报告  */
    short_irq=probe_irq_off(mask);

    if (short_irq==0){ /*  没有探测到中断报告？*/
        printk(KERN_INFO "short: no irq reported by probe\n");
        short_irq=-1;
    }
    /*
     *  如果激活了一个以上的中断，结果就是负的。我们将为中断提供服务(除非是 lpt
     *  端口)并且再次进行循环。最多循环 5 次，然后放弃
     */
} while (short_irq<0 && count++<5);
```

```
if (short_irq<0)
    printk("short: probe failed %i times, giving up\n",count);
```

探测很耗时。尽管 *short* 的探测很快，但象探测帧捕捉卡，就至少需要延迟 20ms(相对处理器时间就太长了)，而探测其它设备可能会更花时间。因此，最好就只在模块初始化时探测中断信号线一次，不管你是在打开设备时(你应该这样做)或者在 *init_module* 中(你无论如何不应该这样做)安装你的中断处理程序的。

值得注意的是，在 Sparc 和 M68k 上，中断探测全无必要，因此也不必实现。探测是种“黑客”行为，象 PCI 这样的成熟的体系结构会提供所有必要的信息。实际上，M68k 和 Sparc 的内核开放给模块桩(stub)的探测函数总是返回 0——每种体系结构都必须定义这些函数，因为它们是由体系结构无关的源文件来开放的。所有其它的体系结构都允许使用上面给出的探测技术。

probe_irq_on 和 *probe_irq_off* 的问题是早期的内核版本并不开放这两个函数。因此，如果你希望写的模块能移植到 1.2 版的内核，你必须自己做中断探测。

DIY(Do It Yourself 自己做)检测

探测也可以有驱动程序自己较容易地实现。如果装载是指定 **probe=2**，*short* 模块将对中断信号线进行 DIY 检测。

实现机制和前面讨论的内核帮助下的检测是一样的：启动所有未被占用的中断，然后等着看会发生些什么。但我们可以利用拥有的对设备的一些知识。通常一个设备可以配置成使用 3 或 4 个中断号中的一个；只需要探测这些中断号，这使我们不必测试所有可能的中断号就可以检测到正确的中断号。

在 *short* 的实现中假定可能的中断号只有 3，5，7 和 9。这些数值实际上是一些并口允许你选取的值的范围。

下面的代码通过测试所有“可能的”中断和会观察发生什么来进行中断探测。**trials** 数组列出所有要尝试的中断号，0 是该列表的结束标志；**trials** 数组用于记录实际上哪个处理程序被驱动程序注册了。

```
int trials[]={3,5,7,9,0};
int tried[]={0,0,0,0,0};
int i,count=0;

/*
*为所有可能的中断信号线安装探测处理程序。记录下结果(0 表示成功，-EBUSY
*表示失败)以便只释放申请的中断
*/
for (i=0; trials[i]; i++)
    tried[i]=request_irq(trials[i], short_probing, SA_INTERRUPT, "short probe", NULL);
```

```
do {
        short_irq=0; /* 尚未取得中断号 */
        outb_p(0x10, short_base+2); /* 启动 */
        outb_p(0x00, short_base);
        outb_p(0xFF, short_base); /* 置位 */
        outb_p(0x10, short_base+2); /* 关闭 */

        /* 处理程序已经设置了这个值 */
        if    (short_irq==0) { /*
              printk(KERN_INFO "short: no irq reported by probe\n");
        }
        /*
        * 如果激活了一个以上的中断，结果就是负的。我们将为中断提供服务(除非是 lpt
        * 端口)并且再次进行循环。最多这样做 5 次
        */
} while(short_irq<=0 && count++<5);

/* 循环结束，卸载处理程序 */
for (i=0; trials[i]; i++)
        if (tried[i]==0)
              free_irq(trials[i],NULL);

if (short_irq<0)
        printk("short: probe failed %i times, giving up\n",count);
```

你可能事先不知道“可能的”中断号。此时，你需要探测所有空闲的中断，而不仅是一些 **trials[]**。为了探测所有的中断，你不得不从 0 号中断探测到 **NR_IRQS-1** 号中断，**NR_IRQS** 是在头文件<**asm/irq.h**>中定义的与平台无关的常数。

现在缺的就是探测处理程序自己了。该处理程序的功能就是根据实际接收到的中断号来更新 **short_irq** 变量。**short_irq** 值为 0 意味着“什么也没有”，而负值意味着存在“二义性”。我选取这些值是为了和 *probe_irq_off* 保持一致，并可以在 *short.c* 中使用同样的代码来调用任何一种探测方法。

```
void short_probing(int irq, void *dev_id, struct pt_regs *regs)
{
        if (short_irq == 0) short_irq = irq;        /* 找到 */
        if (short_irq != irq) short_irq = -irq; /* 有二义性 */
}
```

处理程序的参数稍后会介绍。知道参数 **irq** 是要处理的中断号就足以理解上面的函数了。

快速和慢速中断处理

你已经看到，我为 short 的中断处理程序设置了 **SA_INTERRUPT** 标志位，因此是请求安装一个快速中断处理程序。现在到解释什么是“ 快速 ”和“ 慢速 ”的时候了。实际上，不是所有的体系结构都支持快速和慢速中断处理程序两种实现的。例如，Alpha 和 Sparc 的移植版本，快速和慢速处理程序是一样处理的。2.1.37 版和其后的 Intel 移植版本也消除了两者的差别，因为现代处理器的可以获得的处理能力使得我们不必再区分出快速和慢速两种中断。

这两种中断处理程序的主要差别就在于，快速中断处理程序保证中断的原子处理，而慢速中断处理程序则不保证(这种差别在最新的中断处理的实现也保留了)。也就是说，“ 开启中断 ”处理器标志位(IF)在运行快速中断处理程序时是关闭的，因此在服务该中断时不允许被中断。而调用慢速中断处理时，内核启动微处理器的中断报告，因此在运行慢速中断处理程序时其它中断仍可以得到服务。

在调用实际的中断处理程序之前，不管是快速还是慢速中断处理程序，内核都要执行一项任务，关闭刚才发出报告的那个中断信号线。这对程序员是个好消息－中断服务例程不必是可重入的。但另一方面，即使是慢速中断处理程序也要实现得运行的尽可能快，以免丢失后面到达的中断。

当处理程序还在处理上一个中断时，如果设备又发出新的中断，新的中断会永远丢失。中断控制器并不缓存被屏蔽的中断，但是处理器会进行缓存－一旦发出 *sti* 指令，待处理的中断就会得到服务。*sti* 函数是“ 置中断标志位 ”处理器指令(是在第 2 章“ *编写和运行模块* ”的“ ISA 内存 ”一节引入的)。

总结快速和慢速两种执行环境如下：

- 快速中断处理程序运行时微处理器关闭了中断报告，中断控制器禁止了被服务这个中断。但处理程序可以通过调用 *sti* 来启动处理器的中断报告。

- 慢速处理程序运行时启动了处理器的中断报告，但中断控制器也禁止了被服务这个中断。

但快速和慢速中断处理程序还有另一处不同：内核带来的额外开销。慢速中断处理程序*之所以*慢是因为内核带来的一些管理开销造成的。这意味着较频繁的中断最好由快速中断处理程序为之提供服务。至于 *short*，当把大文件拷贝到*/dev/short0* 时每秒会产生上千次中断。因此我选择使用了一个快速中断处理程序来控制添加给系统的开销。这种分别在更新的 2.1 版的内核中已经得到统一；这个开销现在加到了所有的中断处理程序上。

帧捕捉卡是使用慢速中断处理程序的一个好的候选者。它每秒只中断处理器 50 到 60 次，选择使用慢速处理程序将帧数据从接口卡拷贝到物理内存就不会阻塞住其它的系统中断，例如那些由串口或定时器服务产生的中断。

x86 平台上中断处理的内幕

下面的描述是根据 2.0.x 版本的内核中的两个文件 *arch/i386/kernel/irq.c* 和 *include/asm-i386/irq.h* 推断的；虽然基本概念是相同的，但是具体的硬件细节与平台有关，并且在 2.1 开发版本中有些修改。

最底层的中断处理是在头文件 *irq.h* 中的声明为宏的一些汇编代码，这些宏在文件 *irq.h* 中被扩展。为每个中断声明了三种处理函数：慢速，快速和伪(bad)处理函数。

“伪”处理程序，它最小，是当没有为中断安装C语言的处理程序时的汇编入口点。它将中断转交给适当的PIC(Programmable Interrupt Controller，可编程的中断控制器)设备[*]的同时禁止它，以避免由于伪中断而进一步浪费处理器时间。在驱动程序处理完中断信号后调用 *free_irq*时又会重新安装伪处理程序。伪处理程序不会将*/proc/stat*中的计数器加 1。

值得注意的是，在 x86 和 Alpha 上的自动探测都是依赖于伪处理程序的这种行为。*probe_irq_on* 启动所有的伪中断，而不安装处理程序；*probe_irq_off* 只是简单地检查自调用 *probe_irq_on* 以来那些中断被禁止了。如果你想验证这一点，可以在装载 *short* 时指定 **probe=1**(内核帮助下的检测)，此时可观察到中断计数器没有加 1，而如果装载时指定 **probe=2**(DIY 检测)则会将它们加 1。

慢速中断的汇编入口点会将所有寄存器保存到堆栈中，并将数据段(DS 和 ES 处理器寄存器)指向核心地址空间(处理器已经设置了 CS 寄存器)。然后代码将将中断转交给 PIC，禁止在相同的中断信号线上触发新的中断，并发出一条 sti 指令(set interrupt flag，置中断标志位)。注意处理器在对中断进行服务时会自动清除该标志位。接着慢速中断处理程序就将中断号和指向处理器寄存器的一个指针传递给 *do_IRQ*，这是一个 C 函数，由它来调用相应的 C 语言处理程序。驱动程序传递给中断处理程序参数 **struct pt_regs ***是一个指向存放着各个寄存器的堆栈的指针。

do_IRQ 结束后，会发出 *cli* 指令，打开 PIC 中指定的中断，并调用 *ret_from_sys_call*。最后这个入口点(*arch/i386/kernel/entry.S*)从堆栈中恢复所有的寄存器，处理所有待处理的下半部处理程序(参见本章的“下半部”一节)，并且，如果需要的话，重新调度处理器。

快入口点不同的是，在跳转到 C 代码之前并不调用 *sti* 指令，并且在调用 *do_fast_IRQ* 前并不保存所有的机器寄存器。当驱动程序中的处理程序被调用时，**regs** 参数是 **NULL**(空指针，因为寄存器没有保存到堆栈中)并且中断仍被屏蔽。

最后，快速中断处理程序会重新打开 8259 芯片上的所有中断，恢复先前保存的所有寄存器，并且不经过 *ret_from_sys_call* 就返回了。待处理的下半部处理程序也不运行。

2.1.34 前的所有内核版本中，这两种处理程序在将控制转移给 C 代码前都会将 **intr_count** 变量加 1(参见第 6 章“*时间流*”的“任务队列的特性”一节)。

[*] PC机通常就已经有了两个中断控制芯片，叫做 8259 芯片(主从片)。而可编程的中断控制器设备已经不存在了，但现代的芯片组中也实现了相同的功能。

实现中断处理程序

至此，我们学习了如何注册一个中断处理程序，但还并没有真正编写这样的一个处理程序。实际上，处理程序并没有什么特别的－就是普通的 C 代码。

唯一特别的地方就是处理程序是在中断时间内运行的，因此它的行为要受些限制。这些限制和我们在任务队列中看到的差不多。处理程序不能向用户空间发送或接受数据，因为它不在任何进程的上下文中执行。快速中断处理程序，可以认为是原子地执行的，当访问共享的数据项时并不需要避免竞争条件。而慢速处理程序不是原子的，因为在运行慢速处理程序时也能为其它处理程序提供服务。

中断处理程序的功能就是将有关中断接收的信息反馈给设备，并根据要服务的中断的不同含义相应地对数据进行读写。第一步通常要先清除接口卡上的一个位；大部分硬件设备在它们的“ 中断待处理 ”位被清除前是不会产生任何中断的。一些设备就不需要这一步，因为它们没有“ 中断待处理 ”位；这样的设备比较少，但并口却是其中之一。因此，*short* 不需要清除这样的位。

中断处理程序的典型任务是唤醒在设备上睡眠的那些进程——如果中断向这些进程发出了信号，指示它们等待的事件已经发生，比如，新数据到达了。

还举老的帧捕获卡的例子，进程可以通过连续地对设备读来获取一系列的图像；每读一帧后 *read* 调用都被阻塞，而新的帧一到达中断处理程序都会唤醒该进程。这假定了捕获卡会中断处理器来发出信号通知每一帧的成功到达。

不论是快速还是慢速中断处理程序，程序员都要注意处理例程的执行时间必须尽可能短。如果要进行长时间的计算，最好的方法是使用任务队列，将计算调度到安全时间内进行(参见第 6 章的“ 任务队列 ”一节)。这也是需要下半部处理的一个原因(参见本章稍后的“ 下半部 ”)。

short 中的范例代码使用中断来调用 *do_gettimeofday* 并把当前时间打印到大小为一页的循环缓冲区。然后它唤醒所有的读进程(实际上由于 *short* 使用快速中断处理程序，这些读进程只会在下一个慢速中断处理程序结束时或下一个时钟滴答时醒来)。

```
void short_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    struct timeval tv;
    do_gettimeofday(&tv);

    /* 写一个 16 个字节的记录。假设 PAGE_SIZE 是 16 的倍数 */
    short_head += sprintf((char *)short_head,"%08u.%06u\n",
                          (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
    if (short_head == short_buffer + PAGE_SIZE)
        short_head = short_buffer; /* 绕回来 */
```

```
    wake_up_interruptible(&short_queue); /* 唤醒所有的读进程 */
}
```

这段代码，尽管简单，却给出了一个中断处理程序的典型工作流程。

用来读取在中断时间里填满的缓冲区的节点是*/dev/shortint*。这是唯一的没有在第 8 章中介绍的 *short* 设备节点。*/dev/shortint* 内部的实现为中断产生和报告作了特别的处理。每向设备写入一个字节都会产生一个中断；而读设备时则给出每次中断报告的时间。

如果你将并口插座的第 9 和第 10 引脚相连，那么拉高并行数据字节的最高位就可以产生中断。这可以通过向*/dev/short0* 写二进制数据或者向*/dv/shortint**写入任意数据来实现。

下面的代码是*/dev/shortint* 的 *read* 和 *write* 的实现：

```
read_write_t short_i_read (struct inode *inode, struct file *filp,
                           char *buf, count_t count)
{
    int count0;

    while (short_head == short_tail) {
        interruptible_sleep_on(&short_queue);
        if (current->signal & ~current->blocked) /* 有信号到达 */
          return -ERESTARTSYS; /* 通知 fs 层去处理它 */
        /* 否则，再次循环 */
    }
    /* count0 是可以读进来的数据字节个数 */
    count0 = short_head - short_tail;
    if (count0 < 0) /* wrapped */
        count0 = short_buffer + PAGE_SIZE - short_tail;
    if (count0 < count) count = count0;

    memcpy_tofs(buf, (char *)short_tail, count);
    short_tail += count;
    if (short_tail == short_buffer + PAGE_SIZE)
        short_tail = short_buffer;
    return count;
}

read_write_t short_i_write (struct inode *inode, struct file *filp,
                            const char *buf, count_t count)
{
    int written = 0, odd = filp->f_pos & 1;
    unsigned port = short_base; /* 输出到并口数据锁存器 */
```

```
        while (written < count)
                outb(0xff * ((++written + odd) & 1), port);

        filp->f_pos += count;
        return written;
}
```

使用参数

虽然 *short* 中不对参数进行处理，但还是有三个参数被传给了中断处理函数：**irq**，**dev_id** 和 **regs**。下面我们看看每个参数的意义。

当用一个处理程序来同时对若干个设备进行处理并且使用不同的中断信号线，那么中断号**(int irq)**就很有用了。例如，立体视频系统就使用了两个中断来支持两个帧捕捉卡。驱动程序必须能检测两个设备，并且安装一个处理程序来对两个中断进行处理。驱动程序就可以使用 **irq** 参数来通知处理程序是哪个设备发出了中断。

例如，如果驱动程序声明了一个设备结构的数组 **hwinfo**，每个元素都有一个 **irq** 域，那么下面的代码可以在中断到达时选取出正确的设备。这段代码的设备前缀是 **cx**。

```
static void cx_interrupt(int irq)
{
        /* “ Cxg_Board ”是硬件信息的数据类型 */
        Cxg_Board *board; int i;

        for (i=0, board=hwinfo; i>cxg_boards; board++,i++)
                if    (board->irq==irq)
                        break;

        /* 现在'board' 指向了正确的硬件描述 */
        /* .... */
}
```

第二个参数，**void *dev_id**，是一种 ClientData；是传递给 *request_irq* 函数的一个 **void *** 类型的指针，并且当中断发生时这个设备 ID 还会作为参数传回给处理程序。参数 **dev_id** 是在 1.3.70 版的 Linux 中引入以处理共享中断，但即使不共享它也很有用。

假定我们例子中的设备是象下面这样注册它的中断的(这里 **board->irq** 是要申请的中断，**board** 是 ClientData)

```
static void cx_open(struct inode *inode, struct file *filp)
{
```

---

[*] *shortint*设备是通过交替地向并口写入 0x00 和 0xff来实现的。

```
    Cxg_Board *board=hwinfo+MINOR(inode->i_rdev);
    Request_irq(board->irq, cx_interrupt, 0, "cx100", board /* dev_id */);
    /* .... */
    return 0;
}
```

这样处理程序的代码就可以缩减如下：

```
static void cx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    Cxg_Board *board=dev_id;

    /*  现在'board'  指向了正确的硬件项  */
    /* .... */
}
```

最后一个参数，**struct pt_regs *regs**，很少使用。它存放着在处理器进入中断代码前的一个处理器上下文的快照。这些寄存器可用于监控和调试，实际上 *show_regs* 函数(它是按下 **RightAlt-PrScr** 键时由键盘中断启动的调试函数－第 4 章“ *调试技术* ”的“ 系统挂起 ”一节)就是使用它们来实现监控和调试的。

打开和禁止中断

有时驱动程序要打开和禁止它相应 IRQ 信号的中断报告。内核为此提供了两个函数，都在头文件<**asm/irq.h**>中声明：

```
void disable_irq(int irq);
void enable_irq(int irq);
```

调用其中任一函数都会更新 PIC 中对指定的 **irq** 的掩码。

实际上，当中断被禁止后，那么即使硬件急需处理，处理器也得不到报告。例如，“ x86 上中断处理的内幕 ”一节中就介绍了“ 伪 ”处理程序在 x86 上的实现就禁止了它收到的所有中断。

但是，为什么我们要禁止中断呢？还是举并口的例子，我们看看 *plip*(并行 IP)网络接口。*plip* 设备使用裸的(bare-bones)并口来传输数据。因为只能从并口读出 5 个位，它们就被解释为四个数据位和一个时钟/握手信号。当发起者(即发送数据包的那个接口)送出数据包的第一个位时，时钟信号会升高，接收方接口就会中断处理器。然后 *plip* 处理程序被调用来处理新到达的数据。

在设备被激活后，开始数据传输，使用握手信号将新数据按时钟周期传送给接收接口(这可能不是最好的实现方法，但只有这样才能和其它使用并口的数据包驱动程序兼容)。如果接收接口为接收每个字节(8 个位)都要处理两次中断，那性能必然不可忍受。因此驱动程序

在接收数据包时要禁止中断。

同样的，因为从接收方到发送方的握手信号用于确认数据的接收，发送接口也要在发送数据包时禁止它的中断信号。

但要注意的是，因为处理程序本身无法打开和禁止中断信号。存在这个限制是因为，如上所述，内核在调用处理程序前会禁止中断，而在处理程序结束后又会重新打开它。但打开和禁止中断仍可以做到，只要在下半部处理程序中作就可以了(参见下一节)。

最后值得注意的是，在 Sparc 实现中，*disable_irq* 和 *enable_irq* 都被定义为指针而不是函数。这个小技巧允许内核在启动检测你是在运行哪种 Sparc 时对指针进行相应的赋值(Sun4c 和 Sun4m 的中断硬件不相同)。而所有的 Linux 系统上，不管使不使用这种小技巧，函数在 C 语言中的语义都相同，这就避免了编写那些冗长无味的条件编译代码。

下半部

中断处理的一个主要问题是如何在处理程序中完成比较耗时的任务。Linux 解决这个问题的方法是将中断处理程序划分成两个部分：所谓的“ 上半部 ”是你通过 *request_irq* 函数注册的处理例程，而 “ 下半部 ” (bottom half，简称为 “ bh ”)则是由上半部调度到以后在更安全的时间内执行的那部分例程。

但是下半部有什么用呢？

上半部和下半部处理程序最大的不同就在于在执行 bh 是所有的中断都是打开的－所以说它是在“ 更安全 ”时间内运行。典型的情况是，上半部处理程序将设备数据存放进一个设备指定的缓冲区，再标记它的下半部，然后退出；这样处理得就非常快。由 bh 将新到的数据再分派给各个进程，必要时再唤醒它们。这种设置允许上半部处理程序在下半部还在运行时就能为新的中断提供服务。但另一方面，在上半部处理程序结束前如果有新的数据到了，由于中断控制器禁止了中断信号，这些数据仍会丢失掉。

所有实际的中断处理程序都作了这样的划分。例如，当网络接口卡报告新的数据包到达了，处理程序只是取得数据并将它推进协议层中；对数据包的实际处理是在下半部中完成的。

这使我们想起了任务队列；实际上，任务队列就是从下半部的一个较老的实现演变而来的。甚至 1.0 版的内核也有下半部，而任务队列则还未引入。

与动态的任务队列不同，下半部的个数有限，并由内核预定义了；这和老的内核定时器有些类似。下半部的静态特性并不是个问题，因为有些下半部可以通过运行任务队列演变为动态对象。在头文件<**linux/interrupt.h**>中，你可以看到下半部的一张列表；它们的最有意思的一部分将在下面讨论。

下半部的设计

下半部由一个函数指针数组和一个位掩码组成－这就是为什么它们不超过 32 个的原因。当内核准备处理异步事件时，它就调用 *do_bottom_half*。我们已经在前面看到，从系统调用返回和退出慢速处理程序时，内核都是这样做的；而这两类事件都发生得很频繁。而决定使用掩码主要出于性能的考虑：检查掩码只要一条机器指令，开销最小。

当某段代码需要调度运行下半部处理时，只要调用 *mark_bh* 即可，该函数设置了掩码变量的一个位以将相应的函数进入执行队列。下半部可以由中断处理程序或其它函数来调度。执行下半部时，它会自动去除标记。

标记下半部的函数是在头文件<**linux/interrupt.h**>中定义的：

```
void mark_bh(int nr);
```

这里，**nr** 是激活的 bh 的“数目”。这个数是在头文件<**linux/interupt.h**>中定义的一个符号常数，它标记位掩码中要设置哪个位。每个下半部 bh 相应的处理函数由拥有它的那个驱动程序提供。例如，当调用 **mark_bh(KEYBOARD_BH)**时，要调度执行的函数是 *kbd_bh*，它是键盘驱动程序的一部分。

因为下半部是静态对象，模块化的驱动程序无法注册*自己的*下半部。目前还不支持下半部的动态分配，可能将来也不会支持，因为此时可以使用立即队列。

本节其余部分将列出一些有意思的下半部：

IMMEDIATE_BH
对设备驱动程序来说这是最重要的 bh。被调度执行的函数处理任务队列 **tq_immediate**。没有下半部的驱动程序(例如一个定制模块)可以通过使用立即队列来取得和立即 bh 同样的效果。将任务等记到队列中后，驱动程序必须标记 bh 以使得它的代码真正得到执行；具体做法可参见第 6 章的“立即队列”一节。

TQUEUE_BH
*如果*任务等记在 tq_timer 队列中，那么每次时钟滴答都会激活这个 bh。实际上，驱动程序可以使用 tq_timer 来实现自己的下半部；定时器队列是在第 6 章(“定时器队列”一节中)引入的一种下半部，但并不必为它调用 *mark_bh*。**TQUEUE_BH** 总是在 **IMMEDIATE_BH** *后*执行的。

NET_BH
网络驱动程序通过标记这个队列来将事件通知上面的网络层。bh 本身是网络层的一部分，模块无法访问。我们将在第 14 章“*网络设备驱动程序*”的“中断驱动的操作”一节中熟悉它的使用。

CONSOLE_BH

控制台是在下半部中进行终端tty切换的。这个操作要包含进程控制。例如，在X Window系统和字符模式间切换就是由X 服务器控制的。而且，如果键盘驱动程序请求控制台的切换，那么控制台切换不能在中断时进行。也不能在进程向控制台写的时候进行。使用bh就能满足这些要求，因为驱动程序可以任意禁止下半部；如果发生了前面情况，在写控制台时禁止**console_bh**即可*。

TIMER_BH

这个 bh 由 *do_timer* 函数标记；*do_timer* 函数管理着时钟滴答。这个 bh 要执行的函数正是驱动内核定时器的那个函数。因此不使用 *add_timer* 的驱动程序是无法使用这种功能的。

其余的下半部是有特定的内核驱动程序使用的。没有为模块提供入口点，即使有入口也没什么意义。

bh 一旦被激活，当在 *return_from_sys_call* 中调用 *do_bottom_half*(*kernel/softirq.c*)时它就会得到执行。当进程退出系统调用或慢速中断处理程序退出时都会执行 *return_from_sys_call* 过程。快速中断处理程序退出时就不会执行下半部 如果驱动程序需要快速执行它的下半部。它必须注册一个慢速处理程序。

时钟滴答总要执行 *ret_from_sys_call* 的；因此，如果快速中断处理程序标记了一个 bh，实际的处理函数最多 10ms 后就会被执行(Alpha 上则小于 1ms，它时钟滴答的频率是 1024Hz)。

下半部运行后，如果设置了 **need_resched** 变量，就会调用调度器；各种 *wake_up* 函数都会设置这个变量。因此，上半部可以将任何与被唤醒的进程有关的任务放到下半部去做－这些任务马上就会被调度。例如，当 *telnet* 数据包到达网络时就是这样的。*net_bh* 唤醒 *telnetd*，并且调度器马上给它处理器时间，因此没有额外的延迟。

编写下半部

下半部代码是在安全时间内运行的——比上半部处理程序运行时安全。但是，也有些注意事项，因为 bh 还是在“ 中断时间 ”内处理的。intr_count 不为 0，因为下半部是在进程上下文之外执行的。因此，第 6 章的“ 任务队列的特性 ”一节中列出的各种限制也适用于在下半部中执行的代码。

下半部的主要问题是它们通常要与上半部中断处理程序共享数据结构，因此要避免竞争条件。这意味着要暂时禁止中断或者使用锁的技术。

从前面“ 下半部的设计 ”一节中给出的下半部列表可以明显看出，新编写的实现了下半部的驱动程序应该通过使用立即队列来将它的代码挂在 **IMMEDIATE_BH** 上。如果你的驱动程序很关键，那么甚至可以拥有从内核里分配的 bh 号。但这样的驱动程序比较少，因此我就不详细介绍了。共有三个函数可用于管理自己私有的下半部：*init_bh*，*enable_bh* 和

* 随后就会介绍，使用自己的下半部的驱动程序可以调用*disable_bh*函数。

disable_bh。如果你有兴趣的话，可以在内核源码找到它们。

实际上，使用立即队列和管理自己拥有的下半部并无区别－立即队列也是一种下半部。当标记了 **IMMEDIATE_BH** 后，处理下半部的函数实际上就是去处理立即队列。如果你的中断处理函数将它的 bh 处理函数排进 **tq_immediate** 队列并且标记了下半部，那么队列中的这个任务会正确地被执行。因为所有最新的内核都可以将相同的任务多次排队而不破坏任务队列，因此每次运行上半部处理函数时都可以将下半部排队。稍后我们会看到这种做法。

需要特殊配置的驱动程序——需要多个下半部或不能简单地用 **tq_immediate** 来设置——可以使用定制的任务队列。中断处理函数将任务排进自己的队列中，当它准备运行这些任务时，就将一个简单的对队列进行处理的函数插入立即队列。详情可参见第 6 章的“ 运行自己的任务队列 ”一节。

下面让我们看看 *short* 的实现。装载时如果指定 **bh=1**，那么模块就会安装一个使用了下半部的中断处理函数。

short 是这样对中断处理进行划分的：上半部(中断处理函数)将当前时间保存到一个循环缓冲区中并调度下半部。而 bh 将累积的各个时间值打印到一个字符缓冲区，然后唤醒所有的读进程。

最后上半部非常简单：

```
void short_bh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        do_gettimeofday(tv_head);
        tv_head++;

        if (tv_head == (tv_data + NR_TIMEVAL) )
                tv_head = tv_data; /* wrap */

        /*  将 bh 排队。即使被多次排队也没有关系  */
        queue_task_irq_off(&short_task, &tq_immediate);
        mark_bh(IMMEDIATE_BH);

        short_bh_count++; /*  记录一个新的中断到了  */
}
```

正如我们所料，这段代码调用 *queue_task* 而不会检查任务是否已被排进队列。但在 Linux 1.2 中不能这么做，并且如果你是用 1.2 版的头文件来编译 *short*，那么它会使用不同的处理函数，仅当 **short_bh_count** 为 0 时该函数才会将任务排队。

然后，下半部处理剩下的工作。它也记录下在调度下半部前上半部被激活的次数(savecount)。如果上半部是一个“ 慢速 ”处理函数，那么这个数总为 1，因为如上所述，当慢速处理函数退出时，总会运行待处理的下半部。

```
void short_bottom_half(void *unused)
{
    int savecount = short_bh_count;
    short_bh_count = 0; /*  我们已经从队列中删去*/
    /*
    *  下半部读入由上半部填充的 tv 数组，并将它打印入循环的字符缓冲区，该缓冲区是
    *  由读进程处理的
    */

    /*  首先写入在这个 bh  前发生的中断的次数*/

    short_head += sprintf((char *)short_head,"bh after %6i\n",savecount);
    if (short_head == short_buffer + PAGE_SIZE)
    short_head = short_buffer; /*  绕回来  */

    /*
    *然后，写入时间值。每次写 16 个字节。因此与 PAGE_SIZE 是对齐的
    */

    do {
        short_head += sprintf((char *)short_head,"%08u.%06u\n",
                                    (int)(tv_tail->tv_sec % 100000000),
                                    (int)(tv_tail->tv_usec));
        if (short_head == short_buffer + PAGE_SIZE)
            short_head = short_buffer; /*  绕回来  */

        tv_tail++;
        if (tv_tail == (tv_data + NR_TIMEVAL) )
            tv_tail = tv_data; /*  绕回来  */

    } while (tv_tail != tv_head);

    wake_up_interruptible(&short_queue); /*  唤醒所有读进程  */
}
```

在我的老式的计算上机运行时给出的时间值表明，使用下半部，两个中断间的时间间隔从 53ms 减少到了 27ms，因为上半部处理函数作的工作更少些。但处理中断的总的工作量不变，更快的上半部的优点是禁止中断的时间较短。但这对 *short* 不是个问题，因为只有在中断处理函数结束后才会重新调用产生中断的 *write* 函数(因为 *short* 采用的是快速中断处理函数)，但对真正的硬件中断来说，这个时间还是很有关系的。

下面是当装载 *short* 时指定 **bh=1** 你可能看到的输出结果：

```
morgana%echo 1122334455 > /dev/shortint; cat /dev/shortint
bh after          5
50588804.876653
50588804.876693
50588804.876720
50588804.876747
50588804.876774
```

共享中断

PC 机一个众所周知的“ 特性 ”就是不能将不同的设备挂到同一个中断信号线上。但是，Linux 打破了这一点。甚至我的 ISA 硬件手册—一本没提到 Linux 的书—也说“ 最多只有一个设备 ”可以挂到中断信号线上，除非硬件设备设计的不好，电信号上并无这样的限制。问题在于软件。

Linux 软件对共享的支持是为 PCI 设备做的，但也可用于 ISA 卡。不必说，非 PC 平台和总线也支持共享。

为了开发能处理共享中断信号的驱动程序，必须考虑一些细节。下面会讨论到，使用共享中断的驱动程序不能使用本章描述的一些特性。但最好尽可能对共享中断提供支持，因为这样对最终用户来说比较方便。

安装共享的处理程序

和已经拥有的中断一样，要与它共享的中断也是通过 *request_irq* 函数来安装的，但它们有两处不同：

- 申请共享中断时，必须在 **flags** 参数中指定 **SA_SHIRQ** 位
- **dev_id** 参数*必须*是唯一的。任何指向模块的地址空间的指针都可以，当然 **dev_id** 一定不能设为 **NULL**。

内核为每个中断维护了一张共享处理函数的列表，并且这些处理函数的 **dev_id** 各不相同，就象是驱动程序的签名。如果两个驱动程序都将 **NULL** 注册为它们对同一个中断的签名，那么在卸载时会混淆起来，当中断到达时内核就会出现 oops 消息。我第一次测试共享中断时就发生过这种事情(当时我只是想着“ 一定要将 **SA_SHIRQ** 位加到这两个驱动程序上 ”)。

满足这些条件之后，如果中断信号线空闲或者下面两个条件同时得到满足，那么 *request_irq* 就会成功：

- 前面注册的处理函数的 **flags** 参数指定了 **SA_SHIRQ** 位。

- 新的和老的处理函数同为快速处理函数，或者同为慢速处理函数。

需要满足这些要求的原因很明显：快速和慢速处理函数处于不同的环境，不能互相混淆。类似的，你也不能与已经安装为不共享的中断处理函数共享相同的中断。但关于快速和慢速处理函数的限制对最近的 2.1 版的内核来说是不必要的，因为两种处理函数已经合并了。

当两个或两个以上的驱动程序共享同一根中断信号线，而硬件又通过这根信号线中断了处理器时，内核激活这个中断注册的所有处理函数，并将自己的 **dev_id** 传递给它们。因此，共享处理函数必须能够识别出它对应于哪个中断。

如果你在申请中断信号之前需要探测你的设备的话，内核无法提供帮助。没有共享中断的探测函数。仅当使用的中断信号线空闲时，标准的探测机制才能奏效；但如果被其它的具有共享特性的驱动程序占用的话，那么即使你的程序已经可以正常工作了，探测也会失败。

那么，唯一的可以用来探测共享中断信号的技术就是 DIY 探测。驱动程序必须为所有可能的中断信号线申请共享处理函数，然后观察中断在何处报告。这里和前面介绍的 DIY 的探测之间的差别在于，此时探测处理函数必须检查是否真的发生了中断，因为为响应共享中断信号线上的其它设备的中断它可能已经被调用过了。

释放处理函数同样是通过执行 *release_irq* 来实现的。这里 **dev_id** 参数用于从该中断的共享处理函数列表中正确地选出要释放的那个处理函数。这就是 **dev_id** 指针必须唯一的原因。

使用共享处理程序的驱动程序时还要小心：不能使用 *enable_irq* 和 *disable_irq*。如果它使用了这两个函数，共享中断信号线的其它设备就无法正常工作了。一般地，程序员必须牢记他的驱动程序并不独占这个中断，因此它的行为必须比独占中断信号线时更“社会化”些。

运行处理函数

如上所述，当内核接收到中断时，所有注册过的处理函数都会被激活。共享中断处理程序必须能将需要处理的中断和其它设备产生的中断区分开来。

装载 *short* 时指定 **shared=1** 将安装下面的处理程序而不是缺省的处理程序：

```
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
    int value;
    struct timeval tv;

    /* 如果不是 short，立即返回 */
    value = inb(short_base);
    if (!(value & 0x80)) return;
```

```
        /* 清除中断位 */
        outb(value & 0x7F, short_base);

        /* 其余不变 */

        do_gettimeofday(&tv);
        short_head += sprintf((char *)short_head,"%08u.%06u\n",
                                    (int)(tv.tv_sec % 100000000), (int)(tv.tv_usec));
        if (short_head == short_buffer + PAGE_SIZE)
            short_head = short_buffer; /* 绕回来 */

        wake_up_interruptible(&short_queue); /* 唤醒所有读进程 */
    }
```

解释如下。因为并口没有 “ 待处理的中断 ” 位可供检查，为此处理函数使用了 **ACK** 位。如果该位为高，报告的中断就是送给 *short* 的，并且处理函数将清除该位。

处理函数是通过将并口的数据端口的高位清零来清除中断位的－*short* 假定并口的 9 和 10 引脚是连在一起的。如果与 short 共享同一中断的设备产生了一个中断，*short* 会知道它的信号线并未激活，因此什么也不会做。

显然，*真正的*驱动程序做的工作会更多些；特别的，它要使用 **dev_id** 参数来得到自己的硬件结构。

特性完全的驱动程序可能会将工作划分为上半部和下半部，但这很容易添加，对实现共享的代码并无太大影响。

/proc 接口

系统中安装的共享中断处理程序不会影响*/proc/stat* 文件(该文件甚至并不知道处理程序的存在)。但是，*/proc/interrupts* 文件会有些变化。

为同一个中断号安装的处理程序会出现在*/proc/interrupts* 文件的同一行上。下面的快照取自我的计算机，是在我将 *short* 和我的帧捕捉卡装载为共享中断处理程序之后：

```
 0:    1153617    timer
 1:      13637    keyboard
 2:          0    cascade
 3:      14697 +  serial
 5:     190762    NE2000
 7:       2094 +  short, + cx100
13:          0    math error
14:      47995 +  ide0
15:      12207 +  ide1
```

这里共享中断信号是 IRQ7 号中断；激活的处理程序列在同一行，用逗号隔开。显然内核是无法区分 *short* 中断和捕捉卡(*cx100*)中断的。

中断驱动的 I/O

如果和处理的硬件间的数据传输因为某些原因会被延迟的话，那么驱动程序的写函数必须实现缓冲。数据缓冲可以将数据的发送和接收与 *write* 及 *read* 系统调用分离开来，提高系统的整体性能。

一个好的缓冲机制是“ 中断驱动的 I/O ”，它在中断时间内填充一个输入缓冲区并由读设备的进程将其取空；或由写设备的进程来填充一个输入缓冲区并在中断时间内将其取空。

中断驱动的数据传输要正确进行，要求硬件必须安下面的语义产生中断：

- 对输入而言，当新数据到达，系统处理器准备读取它时，设备就中断处理器。实际执行的动作取决于设备是否使用了 I/O 端口，内存映射或者 DMA。
- 对输出而言，当设备准备好接收新数据或对成功的数据传输进行确认时都会发出中断。内存映射和能进行 DMA 的设备通常是通过产生中断来通知系统它们的对缓冲区的处理已经结束。

read 或 *write* 调用时间和实际的数据到达时间之间的关系是在第 5 章“ *字符设备驱动程序的扩展操作* ”的“ 阻塞型和非阻塞型操作 ”一节中介绍的。中断驱动的 I/O 引入了共享数据项的并发进程间的同步问题，因此所有这些问题都与竞争条件有关。

竞争条件

当变量或其它数据项在中断时间内被修改时，由于竞争条件的存在，驱动程序的操作就有可能造成它们的不一致。当操作不是原子地执行时，竞争条件就会发生，但在执行时仍假定数据会保持一致性。因此“ 竞争 ”是在非原子性的操作和其它可能被同时执行的代码之间发生的。典型的，竞争条件会在三种情况下发生：在函数内隐式地调用 *schedule*，阻塞操作和由中断代码或系统调用访问共享数据。最后一种情况发生得最频繁，因此我们在这一章处理竞争条件。

处理竞争条件是编程时最麻烦的一部分，因为相关的臭虫满足的条件很苛刻，不容易再现，很难分辨出中断代码和驱动程序的方法间是否存在竞争条件。程序员必须极为小心地避免数据或元数据的冲突。

一般用于避免竞争条件的技术是在驱动程序的方法中实现的，这些方法必须保证当数据项受到没有预料到的修改时得到正确的处理。但另一方面，中断处理函数并不需要特别的处理，因为相对设备的方法，它的操作是原子性的。

可以使用不同的技术来防止数据冲突，我下面将介绍最常用的一些技术。我不给出完整的代码，因为各种情况下最好的实现代码取决于被驱动的设备的操作模式以及程序员的不同爱好。

最常用的防止数据被并发地访问的方法有：

- 使用循环缓冲区和避免使用共享变量。
- 在访问共享变量的方法里暂时禁止中断。
- 使用锁变量，它是原子地增加和减少的。

当访问可能在中断时间内被修改了的变量时，不论你选用的是哪种方法，都必须决定如何进行处理。这样的变量可以声明为 **volatile** 的，来阻止编译器对该值的访问进行优化(例如，它阻止编译器在整个函数的运行期内将这个值放进一个寄存器中)。但是，使用 **volatile** 变量后，编译器产生的代码会很糟糕，因此你可能会转向使用 *cli* 和 *sti*。Linux 实现这些函数时使用了 *gcc* 的制导来保证在中断标志位被修改之前处理器处于安全状态。

使用循环缓冲区

使用循环缓冲区是处理并发访问问题的一种有效方法 :当然最好的处理方法还是不允许并发访问。

循环缓冲区使用了一种被称为“ 生产者和消费者 ”的算法－一个进程将数据放进缓冲区中，另一个则将它取出来。如果只有一个生产者和一个消费者，那就避免了并发访问。在 *short* 模块中有两个生产者和消费者的例子。其中一个情形是，读进程等待消费在中断时间里生产的数据；而另一个情形是，下半部消费上半部生产的数据。

共有两个指针用于对循环缓冲区进行寻址：**head** 和 **tail**。**head** 是数据的写入位置，由数据的生产者更新。数据从 **tail** 处读出，它是由消费者更新的。正如我上面提到的，如果数据是在中断时间内写的，那么多次访问 **head** 多次时就必须小心。你必须将 **head** 定义成 **volatile** 的或者在进入竞争条件前将中断禁止。

循环缓冲区在填满前工作的很好。如果缓冲区满了，就可能出问题，但你可以有多种不同的解决方法可供选择。*short* 中的实现就是简单地丢弃数据；并不检查溢出，如果 **head** 超过了 **tail**，那么整个缓冲区中的数据都丢失了。其它的实现还有丢弃最后那个数据项；覆盖缓冲区的 **tail**，*printk* 是这么实现的(参见第 4 章的“ 消息是如何记录的 ”一节)；或者阻塞生产者，*scullpipe* 是这么实现的；或者分配一个临时的附加的缓冲区作为主力缓冲区的候补。最好的解决方案取决于数据的重要性和其它一些具体情况下的问题，所以我就不在这讨论了。

虽然循环缓冲区看来解决了并发访问的问题，但当 *read* 函数进入睡眠时仍有出现竞争条件的可能。下面的代码给出 *short* 中这个问题出现的位置：

```
while (short_head==short_tail) {
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

执行这个语句时，新数据有可能在**while**条件被测试是否为真*后*和进程进入睡眠*前*到达。中断中携带的信息就无法被进程及时读取；因此即使此时 **head != tail** 进程也将进入睡眠，直到下一项数据到达时它才会被唤醒。

我并没有为 *short* 实现正确的锁，因为 *short_read* 的源码在第 8 章的“ 驱动程序样例 ”一节中就包括了，当时还没有讨论到这一点。而且，short 处理的数据也不值得我们为它这么做。

尽管 *short* 收集的数据并不重要，而且在连续的两条指令时间间隔内发生中断的可能性小到可以忽略，但是有些时候你还是不能在还有待处理的数据时冒险地进入睡眠。

但这个问题一般来说还是值得对它进行特别的处理的，我们将它留到本章后面的“ 无竞争地进入睡眠 ”一节，那里我将会更详细地进行讨论。

值得注意的是，循环缓冲区只能处理生产者和消费者的情形。程序员必须经常地通过更复杂的数据结构来解决并发访问的问题。生产者/消费者的情形实际上是这些问题中最简单的一种；其它的数据结构，比如象链接表，就不能简单地使用循环缓冲区的实现方案。

禁止中断

获得对共享数据独占访问的通用方法是调用 *cli* 来禁止处理器的中断报告。当数据项(例如链接表)在中断时间内要被修改*并且*是被生存于正常的计算流中的函数修改时，那么随后的函数在访问这些数据前就必须先禁止中断。

这种情况下，竞争条件会发生在读共享数据项的指令和使用刚获得与数据有关的信息的指令之间。例如，如果链接表在中断时间内被修改过了，那么下面的循环在读这个表时就可能会失败。

```
for (ptr=listHead; ptr; ptr=ptr->next)
    /* do somthing */;
```

在 **ptr** 已经被读取后但在使用它之前，一个中断可能会改变了 **ptr** 的值。如果发生了这种情况，你一使用 **ptr** 就会有问题，因为这个指针当前的值与链接表已经没有关系了。

一个可能的解决的方法就是在整个关键循环期间都将中断禁止。虽然禁止中断的代码早在第 2 章的“ ISA 内存 ”一节中就已经引入了，但仍值得在这里再重复一遍：

```
unsigned long flags;
save_flags(flags);
```

```
cli();
/* 临界区代码 */
restore_flags(flags);
```

实际上，在驱动程序的方法中，可以就用简单的 *cli/sti* 对来替代，因为你可以认为当进程进入系统调用时中断会被打开。但是，在要被其它代码所调用的代码中，你不得不使用更安全的 *save_flags/restore_flags* 解决方法，因为此时无法确定中断标志位(IF) 当前的值。

使用锁变量

共享数据变量的第三种方法是使用使用原子指令进行访问的锁。当两个无关的实体(比如象中断处理程序和 read 系统调用，或者是 SMP 对称多处理器计算机中的两个处理器)需要并发地对共享的数据项进行访问时，它们必须先申请锁。如果得不到锁，它就必须等待。

Linux 内核开放了两套函数来对锁进行处理：位操作和对“ 原子性 ”数据类型的访问。

位操作

经常的，我们要使有单个位的锁变量或者要在中断时间内更新设备状态位－而进程可能正在访问它们。内核为此提供了一套原子地修改和测试位的函数。因为整个操作是单步完成的，因此不会介入任何中断。

原子性的位操作运行的很快，因为它们通常不禁止中断，使用单条机器指令来完成相应操作。这些函数与体系结构相关，在头文件**<asm/bitops.h>**中声明。即使在 SMP 机器上它们也能保证是原子的，因此是推荐的保持处理器间一致性的方式。

不幸的是，这些函数的数据类型也是体系结构相关的。**nr** 参数和返回值在 Alpha 上是 **unsigned long** 类型，而在其它体系结构上是 **int** 类型。下面的列表描述了 1.2 到 2.1.37 各版的位操作形式。但该列表在 2.1.38 版中有了改变，详情可参见第 17 章“ *近期发展* ”的“ 位操作 ”一节。

set_bit(nr, void *addr);

这个函数用于设置 **addr** 指向的数据项的第 **nr** 个位。该函数作用在一个 **unsigned long** 上，即使 **addr** 指向 **void**。返回的是该位原先的取值－0 或非零。

clear_bit(nr, void *addr);

这个函数用于清除 **addr** 指向的 **unsigned long** 数据中的指定位。它的语义和 *set_bit* 类似。

change_bit(nr, void *addr);

这个函数用于切换指定位，其它方面和前面的 *set_bit* 和 *clear_bit* 函数类似。

test_bit(nr, void *addr);

这个函数是唯一一个不必是原子的为操作；它只是简单地返回该位当前的值。

当这些函数用于访问和修改共享的位时，你只要调用它们即可。而使用位操作来管理控制共享变量访问的锁变量，则更复杂些，需要举一个例子。

要访问共享数据项的代码段可以使用 *set_bit* 或 *clear_bit* 来试着原子地获取锁。通常是象下面的代码段这样实现的；假定锁位于地址 **addr** 的第 **nr** 位上。并且假定当锁空闲时该位为 0，锁忙时该位非零。

```
/* 试着设置锁 */
while (set_bit(nr,addr)!=0)
        wait_for_a_while();

/* 做你的工作 */

/* 释放锁，并检查... */
if (clear_bit(nr,addr)==0)
        something_wnt_wrong(); /* 已经被释放了：出错 */
```

这种访问共享数据的方式的毛病是竞争双方都必须要等待。如果其中一方是中断处理程序，那么这一点就较难保证了。

原子性的整数操作

内核程序员经常需要在中断处理程序和其它函数间共享整数变量。我们刚才已经看到对位的原子访问还不足以保证一切都能运行正常(对前面的例子来说，如果一方是一个中断处理函数的话，那就必须使用 *cli*)。

实际上，防止竞争条件的需要是如此迫切，以致于内核的开发者为这个问题专门实现了一个头文件：**<asm/atomic.h>**。这个头文件比较新，Linux 1.2 中就没有提供。因此，需要向后兼容的驱动程序是不能使用的。

atomic.h 中提供的函数比刚才介绍的那些位操作功能更强大。*atomic.h* 中定义了一种新的数据类型，**atomic_t**，只能通过原子操作来访问它。

atomic_t 目前在所有支持的体系结构上都被定义为 **int**。下面的操作是为这个数据类型所定义的，能保证 SMP 机器上的所有处理器是原子地对它进行访问。这些操作都非常快，因为它们都尽可能编译成单条的机器指令。

void atomic_add(atomic_t i, atomic_t *v);

将 **v** 指向的原子变量加上 **i**。返回值是 **void** 类型，大部分时候没有必要知道新值。网络部分的代码使用这个函数来更新套接字在内存使用上的统计信息。

void atomic_sub(atomic_t i, atomic_t *v);

从**v 里减去 i。在最新的 2.1 版的内核中这两个函数的参数 i 都声明成 int 类型，但这种改变主要是出于美观的需要，并不对源代码造成影响。

void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
对原子变量加减 1。

int atomic_dec_and_test(atomic_t *v);
该函数是在 1.3.84 版的内核里加入的，用于跟踪引用计数。仅当变量**v 在减 1 后取值为 0 时返回值为 0。

如上所述，只能使用上面这些函数来访问 **atomic_t** 类型的数据。如果你将原子数据项传递给了一个要求参数类型为整型的函数，编译时就会得到警告。不用说，可以读取原子数据项的当前值并将它强制转换成其它数据类型。

无竞争地进入睡眠

在讨论进入睡眠的问题中我们曾忽略了一个竞争条件。这个问题实际上要比中断驱动的 I/O 问题更普遍，而有效的解决方案需要对 *sleep_on* 的实现内幕有些了解。

这种特别的竞争条件发生在检查进入睡眠的条件和对 *sleep_on* 的实际调用之间。下面的测试代码和前面使用的代码是一样的，但我觉得还是值得再在这里列出：

```
while (short_head==short_tail){
    interruptible_sleep_on(&short_queue);
    /* ... */
}
```

如果要安全地进行比较和进入睡眠，你必须*先*禁止中断报告，*然后*测试条件并进入睡眠。因此，比较中被测试的变量不会被修改。内核允许进程在发出 *cli* 指令后就进入睡眠。而在将进程插入它的等待队列之后，在调用 *shcedule* 之前，内核只要简单地重新打开中断报告就可以了。

这里给出的例子代码使用了 **while** 循环，由该循环来进行信号处理。如果有阻塞的信号向进程发出报告，*interruptible_sleep_on* 就返回，再次进行 **while** 语句中的测试。

下面是一种可能的实现：

```
while (short_head==short_tail){
    cli();
    if (short_head==short_tail)
        interuptible_sleep_on(&short_queue);
    sti();
    /* ... 信号解码 .... */
```

```
}
```

如果中断是在 *cli* 后发生的，那么这个中断在当前进程进入睡眠前都会处于待处理状态。而当中断最终报告给处理器时，进程已经进入了睡眠，可以被安全地唤醒。

在这个例子中，我可以使用 *cli/sti*，是因为设计的这段范例代码存在于 *read* 方法内的；否则我们必须使用更为安全的 *save_flags*，*cli*，和 *restore_flags* 函数。

如果在进入睡眠之前你不想禁止中断，那么还有另一种方法来完成与上面相同的任务(Linus 非常喜欢用这种方法)。但是，如果你愿意的话你可以跳过下面的讨论，因为下面的讨论的确有点太细了。

该方法的基本想法是，进程可以把自己排进等待队列，声明自己的状态为睡眠状态，*然后*执行它的测试代码。

典型的实现如下：

```
struct wait_queue wait = {current, NULL};

add_wait(&short_queue, &wait);
current->state=TASK_INTERRUPTIBLE;
while (short_head==short_tail){
    schedule();
    /* ... 信号解码 ... */
}
remove_wait_queue(&short_queue, &wait);
```

这段代码看起来有点象将 *sleep_on* 的内部实现展开了。显式地声明了 **wait** 变量，因为需要用它来使进程进入睡眠；这一切是在第 5 章的“ 等待队列 ”一节中解释的，但这个例子中引入了一些新的符号。

current->state

这个字段是给调度器用的提示。调度器被激活后，它将通过观察所有进程的 **state** 字段来决定接着作些什么。所有进程都可以任意修改自己的 **state** 字段，但在调度器运行之前这种改变还不会生效。

#include <linux/sched.h>

TASK_RUNNING

TASK_INTERRUPTIBLE

TASK_UNINTERRUPTIBLE

这些符号名代表了 **current->state** 最经常取的一些值。**TASK_RUNNING** 表示进程正在运行，其它两个表示进程正在睡眠。

void add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

void remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void __add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
void __ remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)
这些函数用于从等待队列中插入和删除进程。wait 参数必须指向进程堆栈所在的页(临时变量)。以下划线开头的函数运行的更快些，但它们在禁止中断后才能被调用(例如，在快速中断处理程序内)。

有了这些背景知识，下面让我们看看当中断到达时会发生什么。此时处理程序将调用 **wake_up_interruptible(&short_queue)** ；对 Linux 而言，这意味着“将 **state** 置为 **TASK_RUNNING** ”。因此，如果在 **while** 条件和 *schedule* 调用间有中断报告的话，该任务的 state 字段将会又被标记为 **TASK_RUNNING** 的，因此不会丢失数据。

而如果进程仍是“可中断的”(**TASK_INTERRUPTIBLE**)，*schedule* 将保持它的睡眠状态。

值得注意的是，*wake_up* 系统调用并不会将进程从等待队列中删去。是由 *sleep_on* 来对等待队列进行进程的添加和删除的。因此程序代码必须显式地调用 *add_wait_queue* 和 *remove_wait_queue*，因为这种情况下不再使用 *sleep_on* 了。

中断处理的版本相关性

不是所有本章引入的代码都能向后兼容地移植到 Linux 1.2 上的。在此我将列出主要的差异并对如何处理这些差异提出建议。实际上，*short* 在 2.0.x 和 1.2.13 版的内核上都编译和运行得很好。

request_irq 函数的不同原型

我在这一整章中使用的给 *request_irq* 函数传递参数的方式都是到 1.3.70 版的内核才引入的，因为是到这个版本才出现了共享中断处理程序的。

更早的内核版本并不需要 **dev_id** 参数，原型也相对简单些：

```
int request_irq(unsigned int irq,
                void (*handler)(int, struct pt_regs *),
                unsigned long flags, const char *device);
```

只要使用下面的宏定义(注意早期的版本中 *free_irq* 也没有 **dev_id** 参数)，新的语义可以很容易地强加在旧原型上：

```
#if LINUX_VERSION_CODE < VERSION_CODE(1,3,70)
    /* 预处理器必须能处理递归的定义 */
#   define request_irq(irq,fun,fla,nam,dev) request_irq(irq,fun,fla,nam)
#   define free_irq(irq,dev)                free_irq(irq)
#endif
```

这些宏只是简单地丢弃额外的 **dev** 参数。

处理函数原型上的差异通过显式的**#if/#else/#endif** 语句得到很好的处理。如果你使用了 **dev_id** 指针，旧内核的条件分支可以将它申明为 **NULL** 变量，这样处理函数体就可以对 **NULL** 设备指针进行处理了。

short 模块中的一个例子可以作为这种想法的范例：

```
#if LINUX_VERSION_CODE < VERSION_CODE(1,3,70)
void short_sh_interrupt(int irq, struct pt_regs *regs)
{
  void *dev_id = NULL;
#else
void short_sh_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
#endif
```

探测中断信号线

内核到 1.3.30 版开始开放探测函数。如果你希望你的驱动程序能移植到旧的内核上，你将不得不实现 DIY 检测。实际上早在 1.2 版的内核，这些函数就已经存在了，只是模块化的驱动程序无法使用罢了。

这样，移植中断处理函数就没有什么其它的问题了。

快速参考

本章引入了下面这些与中断管理有关的符号：

```
#include <linux/sched.h>
int request_irq(unsigned int irq, void (*handler)())；
unsigned long flags, const char *device, void *dev_id);
void free_irq(unsigned int irq, void *dev_id);
```

这些系统调用用于注册和注销中断处理程序。低于 2.0 版的内核不提供 **dev_id** 参数。

SA_INTERRUPT
SA_SHIRQ
SA_SAMPLE_RANDOM

这些是 *request_irq* 函数的各种选项。**SA_INTERRUPT** 请求安装快速中断处理程序(相对于慢速处理函数)。**SA_SHIRQ** 安装共享中断处理函数，而第三种选项表明产生的中断的时间戳对系统熵池(entropy pool)有贡献。

/proc/interrupts
/proc/stat
这些文件系统节点用于报告关于硬件中断和安装的处理函数的信息。

unsigned long probe_irq_on(void);
int probe_irq_off(unsigned long);
当驱动程序需要探测设备使用哪根中断信号线时，可以使用这些函数。在中断产生之后，*probe_irq_on* 的返回值必须传回给 *probe_irq_off*。*probe_irq_off* 的返回值就是检测到的中断号。

void disable_irq(int irq);
void enable_irq(int irq);
驱动程序可以启动和禁止中断报告。禁止中断后，硬件产生的中断都将丢失。在上半部处理程序中调用这些函数则没有任何效果。而使用共享中断处理程序的驱动程序决不能使用这些函数。

#include <linux/interrupt.h>
void mark_bh(int nr);
这些函数用于标记要执行的下半部。

#include <asm/bitops.h>
set_bit(nr, void *addr);
clear_bit(nr, void *addr);
change_bit(nr, void *addr);
test_bit(nr, void *addr);
这些函数用于原子性地访问位的值；它们可作用于标志位和锁变量。使用这些函数避免了所有与对位的并发访问有关的竞争条件。

#include <asm/atomic.h>
typedef int atomic_t;
void atomic_add(atomic_t i, atomic_t *v);
void atomic_sub(atomic_t i, atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
这些函数用于原子地访问整数变量。如果想让编译时不出现警告信息，必须只使用这些函数来访问 **atomic_t** 类型的变量。

#include <linux/sched.h>
TASK_RUNNING
TASK_INTERRUPTIBLE
TASK_UNINTERRUPTIBLE
这些是 **current->state** 最经常取的一些值。它们是给 *schedule* 用的提示。

void add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

void remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

void __add_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

void __ remove_wait_queue(struct wait_queue ** p, struct wait_queue *wait)

这些是使用等待队列的最底层的函数。打头的下划线标志该函数是底层的函数，使用后两个函数时处理器必须已经禁止了中断报告。

第 10 章 合理使用数据类型

在进一步讨论更深的主题之前，我们需要先停一停，快速地回顾一下可移植问题。Linux1.2 版本和 2.0 版本之间的不同就在于额外的多平台能力；结果是，大多数源代码级的移植问题已经被排除了。这意味着一个规范的 Linux 驱动程序也应该是多平台的。

但是，与内核代码相关的一个核心问题是，能够同时存取各种长度已知的数据项(例如，文件系统数据类型或者设备卡上的寄存器)和利用不同处理器的能力(32 位和 64 位的体系结构,也有可能是 16 位的)。

当把 x86 的代码移植到新的体系结构上时,核心开发者遇到的好几个问题都和不正确的数据类型相关。坚持强数据类型以及编译时使用 *-Wall -Wstrict-prototypes* 选项能够防止大部分的臭虫。

内核使用的数据类型划分为三种主要类型：象 **int** 这样的标准 C 语言类型，象 **u32** 这样的确定数据大小的类型和象 **pid_t** 这样的接口特定类型。我们将看一下这三种类型在何时使用和如何使用。本章的最后一节将讨论把驱动器代码从 x86 移植到其它平台上可能碰到的其它一些典型问题。

如果你遵循我提供的这些准则,你的驱动程序甚至可能在那些你未能进行测试的平台上编译并运行。

使用标准 C 类型

大部分程序员习惯于自由的使用诸如 **int** 和 **long** 这样的标准类型，而编写设备驱动程序就必须细心地避免类型冲突和潜在的臭虫。

问题是，当你需要“ 2 个字节填充单位(filler) ”或“ 表示 4 个字节字符串的某个东西 ”时，你不能使用标准类型，因为通常的 C 数据类型在不同的体系结构上所占空间大小并不相同。例如，长整数和指针类型在 Alpha 上和 x86 上所占空间大小就不一样，下面的屏幕快照表明了这一点：

```
morgana% ./datasize
system/machine: Linux i486
sizeof(char) =     1
sizeof(short) =    2
sizeof(int) =      4
sizeof(long) =     4
sizeof(longlong) = 8
sizeof(pointer) =  4
```

```
wolf% ./datasize
system/machine: Linux alpha
sizeof(char) =     1
sizeof(short) =    2
sizeof(int) =      4
sizeof(long) =     8
sizeof(longlong) = 8
sizeof(pointer) =  8

sandra% ./datasize
system/machine: Linux sparc
sizeof(char) =     1
sizeof(short) =    2
sizeof(int) =      4
sizeof(long) =     4
sizeof(longlong) = 8
sizeof(pointer) =  4
```

*datasize*程序是一个可以从在 O'Reilly FTP 站点的 *misc-progs* 目录下获得的小程序。

在混合使用 int 和 long 类型时，你必须小心，有时有很好的理由这样做，一种情形就是内存地址，一涉及到内核，内存地址就变得很特殊。虽然概念上地址是指针，但是通过使用整数类型，可以更好地实现内存管理；内核把物理内存看做一个巨大的数组，内存地址就是这个数组的索引。而且，一个指针很容易被取地址(deference)，而使用整数表示内存地址可以防止它们被取地址，这正是人们所希望的(比使用指针更安全)。因而，内核中的地址属于 **unsigned long** 类型，这是利用了指针和长整数类型大小总是相同这一事实，至少在所有 Linux 当前支持的平台上是这样的。我们等着看看将来把 Linux 移植到不符合这一规则的平台上的时候，会发生些什么。

分配确定的空间大小给数据项

有时内核代码需要指定大小的数据项，或者用来匹配二进制结构*或者用来在结构中插入填充字段对齐数据。

为此目的，内核提供如下的数据类型，它们都在头文件**<asm/types.h>**中声明，这个文件又被头文件**<linux/types.h>**所包含：

```
u8;     /* 无符号字节(8 位) */
u16;    /* 无符号字(16 位) */
u32;    /* 无符号 32 位数值 */
u64;    /* 无符号 64 位数值 */
```

* 读分区表时，执行二进制文件时或者解码一个网络包时，就会发生这种情况。

这些数据类型只能被内核代码所访问(也即,在包含头文件<linux/types.h>之前必须先定义__KERNEL__)。相应的有符号类型也是存在的，但一般不用；如果你需要使用它们的话，只要把名字中的 u 替换为 s 就可以了。

如果用户空间的程序需要使用这些类型，可以在这些名字前面添加 2 个下划线：__u8 和其它类型是独立于__KERNEL__定义的。例如，如果一个驱动程序需要通过 ioctl 系统调用与一个运行在用户空间内的程序交换二进制结构的话，头文件必须将结构中的 32 位字段定义为__u32。

重要的是要记住这些类型特定于 Linux，使用它们就会防碍软件向其他 Unix 变体的移植。但是，有些情况下也需要明确说明数据大小，而标准头文件(在每个 Unix 系统上都能找到的)并未声明较合适的数据类型。

你也许注意到，有时内核也使用一般的数据类型，象unsigned int，用于那些大小与体系结构无关的项。这通常是为了向后兼容。当u32 及其相关类型在 1.1.67 版本引入时开发者没办法把存在的数据类型改成新类型，因为当结构字段和赋予的值之间类型不匹配时，编译器会发出警告[+]。Linus当初可没预料到为自己使用而编写的这个操作系统会发展成为多平台的；因此，一些旧的结构的数据类型定义上不是很严格。

接口特定的类型

内核中最常使用的数据类型有它们自己的 typedef 声明,这样就防止了任何移植上的问题。例如，进程号(pid)通常使用 pid_t，而不是 int。使用 pid_t 屏蔽了任何实际数据类型之间可能的差别。我使用“接口特定”这种表述来指代特定数据项的编程接口。

属于指定“标准”类型的其它数据项也可以认为是接口特定的。比如，一个 jiffy 计数总是属于 unsigned long 类型的，独立于它的实际大小－你喜欢那么频繁地使用 jiffy_t 类型么？这里我关注的是接口特定类型的第一类，那些以_t 结尾的类型。

_t 类型完整的列表在头文件<linux/types.h>中，但是该列表几乎没什么用。当需要一个特定类型时，你可以在你要调用的函数原型或者使用的数据结构中找到它。

只要你的驱动程序使用了需要这种“定制”类型的函数，又不遵循惯例的时候，编译器都会发出一个警告；如果你打开 *-Wall* 编译开关并且细心地去除了所有警告，你就可以自信你的代码是可移植的了。

_t 数据项的主要问题是当你需要打印它们的时候，并不总是容易选择正确的 *printk* 或者 *printf* 格式，并且你在一种体系结构上排除了的警告，在另一种体系结构上可能又会出现。例如,当 size_t 在一些平台上是 unsigned long,而在另外一些平台上却是 unsigned int 时，你怎么打印它呢？

[+] 实际上，即使两种类型仅是同一对象的不同名字，例如PC上的**unsigned long**和**u32** 类型，编译器也会发出类型不匹配的信号。

任何时候，当你需要打印一些特定接口的数据的时候，最行之有效的方法就是，把它强制转换成最可能的类型(通常是 long 或 unsigned long 类型)，然后把它用相应的格式打印出来。这种做法不会产生错误或者警告，因为格式和类型相符，而且你也不会丢失数据位，因为强制类型转换要么是个空操作，要么是将该数据项向更大数据类型的扩展。

实际上，通常我们并不会去打印我们讨论的这些数据项，因此只有显示调试信息时才会碰到这些问题。更经常的，除了把接口特定的类型作为参数传递给库或内核函数以外，代码仅仅只会对它们进行些储存和比较。

虽然大多数情形下，_t 类型都是正确的解决方案，但有时候正确的类型也可能并不存在。这会发生在一些还没被抛弃的旧接口上。

在内核头文件中我发现一处疑点，为 I/O 函数声明数据类型时不是很严格(参见第 8 章*“硬件管理”*中的“平台相关性”一节)。这种不严格的类型定义主要是出于历史上的原因，但在编写代码时却会带来问题。就我而言，我经常在把参数交换给 *out* 函数时遇上麻烦；而如果定义了 **port_t**，编译器将会指出这些错误。

其它与移植有关的问题

除了数据类型定义问题之外，如果想让你编写的驱动程序能在不同的 Linux 平台间移植的话，还必须注意到其它一些软件上的问题:

时间间隔

在处理时间间隔时，不能假定每秒一定有 100 个 jiffy。虽然对当前的 Linux-x86 而言这是对的，但并不是所有 Linux 平台都是以 100**HZ** 运行。如果你改变了 **HZ** 的数值，那么即使对 x86，这种假设也是错误的，何况没人知道未来的内核会发生些什么变化。使用 jiffy 计算时间间隔的时候，应该把时间转换成以 **HZ** 为单位。例如，为了检测半秒钟的超时，可以把消逝的时间和 **HZ/2** 作比较。更常见的，与 **msec** 毫秒对应的 jiffy 的数目总是 **msec*HZ/1000**。许多的网络驱动程序在移植到 Alpha 上时都必须修正该细节；有些开始是为 PC 设计的驱动程序给超时明确定义了一个 jiffy 值，但是 Alpha 却有着不同的 **HZ** 数值。

页大小

使用内存时，要记住内存页的大小为 **PAGE_SIZE** 字节，而不是 4KB。假设页大小就是 4KB 并硬编码该数值是 PC 程序员常犯的错误－Alpha 页大小是这的两倍。相关的宏有 **PAGE_SIZE** 和 **PAGE_SHIFT**。后者包含要得到一个地址所在页的页号时需要对该地址右移的位数。对当前的 4KB 和 8KB 的页，这个数值通常是 12 或者 13。这些宏在头文件<**asm/page.h**>中定义。

让我们来看一种简单的情况。如果驱动程序需要 16KB 空间来存放临时数据，它不应当指定 *get_free_pages* 函数的参数 **order**(“2”的幂)。需要一种可移植的解决办法。此时，可以使用条件编译**#ifdef __alpha__**，但这只适用于已知的平台，而如果要支持别的平台，它就不能奏效了。我建议使用下面的代码：

```
buf = get_free_pages(GFP_KERNEL， 14 - PAGE_SHIFT， 0 /*dma*/)；
```

或者，更好一些的代码：

```
int order = (14 - PAGE_SHIFT >0) ? 14 - PAGE_SHIFT : 0；
buf = get_free_pages(GFP_KERNEL， order， 0 /*dma*/)；
```

两种解决办法都利用了 16KB 等于 **1<<14** 这一常识。两个数的商就是它们对数的差(的幂)，而 **14** 和 **PAGE_SHIFT** 都是幂。第二种解决办法就更好，因为它可以防止把一个负的 order 值传递给 *get_free_pages* 函数；**order** 值时在编译时就计算好的，没有运行时的额外开销，而且，上面给出的实现方法是不依赖于 **PAGE_SIZE** 来分配任何 2 的幂次大小的内存空间的安全方法。

字节序

要小心的是不要主观假设字节序。虽然 PC 是按低字节优先的方式存储多个字节(“小印地安，little endian”)，但是大多数更高级的平台是以另一种方式工作的(“大印地安，big endian”)。虽然好的程序不会依赖于字节序，但有时驱动程序需要创建占一个字节以上的整数，或者相反(一个字节以下)。此时，代码中就应该将头文件<**asm/byteorder.h**>包含进来，并且检测头文件中是否定义了**__BIG_ENDIAN** 或**__LITTLE_ENDIAN**。起始的下划线在 Linux-1.2 之后版本的头文件中却去掉了，在头文件<**asm/byteorder.h**>后再包含 *scull* 示例程序中的头文件 *sysdep.h* 就可以修正这个不兼容。

当字节序相关问题与网络传输有联系的时候，就应当使用下面各种函数来进行 16 位和 32 位数值的转换，这些函数也都是在头文件<**asm/byteorder.h**>中定义的：

```
unsigned long   ntohl(unsigned long)；
unsigned short ntohs(unsigned short)；
unsigned long   htonl(unsigned long)；
unsigned short htons(unsigned short)；
```

在网络程序员当中，这些函数是众所周知的。它们得名于“Network TO Host Long”(从网络到主机的 long 类型)或类似的短语。

2.1.10 版的内核增加了 cpu-to-little-endian 和 cpu-to-big-endian 两种转换，2.1.43 版的内核在这方面又加以扩充。新增的一些实用函数将在第 17 章“*近期发展*”中的“转换函数”一节中描述。

数据对齐

在编写可移植代码时最后一个值得考虑的问题是如何访问未对齐数据－例如，当一个 4 字节的数值被储存在不是 4 字节的整数倍的地址中时，如何将它读出来。PC 的用户常常访问未对齐的数据项，但并不是所有体系结构都允许这样做。举个例子，在 Alpha 上，每当程序试图传送未对齐数据时，都会产生一个异常。如果你需要访问未对齐数据，可以使用下面这些宏：

```
#include <asm/unaligned.h>
get_unaligned(ptr)；
put_unaligned(val，ptr)；
```

这些宏是与类型无关的。对各种数据项，不管它是 1 字节，2 字节，4 字节还是 8 字节，这些宏都有效。在 2.0 版以前的内核并不提供这些宏，但在 1.2 版的内核在头文件 *sysdep.h* 中对它们作了定义。

一个通用准则就对显式的常数值持怀疑态度。通常，使用预编译的宏来使代码参数化使代码更通用。虽然我不能在此列出所有参数化的值，但你可以在头文件中找到正确的提示。

不幸的是，有些地方问题还没有得到解决，例如对磁盘扇区数据的处理。出于历史的原因，Linux 只能处理**.5KB** 的磁盘扇区。所幸的是，现存所有设备都满足这个限制。目前正在逐渐地改进代码以支持不同的扇区大小。但要找到代码中所有在**.5KB** 的假设下进行硬编码的地方却非常困难。扇区大小的问题将在第 12 章“*加载块设备驱动程序*”中进一步描述。

快速参考

在本章引入了如下一些符号：

#include <linux/types.h>
typedef u8；
typedef u16；
typedef u32；
typedef u64；

这些类型保证是 **8-**、**16-**、**32-**和 **64-**位的无符号整数值。对应的有符号类型同样存在。在用户空间，你可以通过**__u8**，**__u16** 等来引用这些类型。

#include <asm/page.h>
PAGE_SIZE
PAGE_SHIFT

这些符号定义了当前体系结构下每页包含的字节数和页偏移量所占位数(12 对应 4KB 的页而 13 对应 8KB 的页)。

#include <asm/byteorder.h>
__LITTLE_ENDIAN
__BIG_ENDIAN

两个符号中只能定义其一，这依赖于体系结构。版本 1.3.18 和更老的版本中也声明了这些符号，但是没有打头的下划线(因而和一些网络部分的头文件会发生冲突)。

#include <asm/byteorder.h>
unsigned long ntohl(unsigned long)；
unsigned short ntohs(unsigned short)；
unsigned long htonl(unsigned long)；

unsigned short htons(unsigned short)；

这些函数在网络字节序和主机字节序间转换 long 类型和 short 类型的数据。

#include <asm/unaligned.h>

get_unaligned(ptr)；

put_unaligned(val，ptr)；

一些体系结构需要使用这些宏来保护对未对齐数据的访问。在这些体系结构上，这些宏扩展为通常的对指针取地址的操作，以允许你访问未对齐的数据。

第 11 章 kerneld 和高级模块化

在本书的第二部分，我们要讨论的话题比到目前为止我们所接触过的话题都更为高级。我们将再次从模块化讲起，第二章“ *编写和运行模块* ”中对模块化的介绍只是其中的一部分；*modules* 包(它们的最新版本被称作 *modutils*)支持一些更高级的特性，它们比前面讨论的安装和运行一个基本的驱动程序所需的特性要更为复杂。

本章将讨论 *kerneld* 程序，模块中的版本支持(一种便利性，它使你在升级内核时不必重新编译你的各个模块)以及在卸载和重新装载一个模块时对数据持久性的支持。最后这项功能只有 2.0.0 版或更新版本的 *modules* 包才提供。

按需加载模块

为了方便用户加载和卸载模块，并且避免把不再使用的驱动程序继续保留在核心中浪费内核存储空间，Linux 提供了对模块的自动加载和卸载的支持。(在 1.2 版以前不提供这种支持)要利用这个特性，在编译内核前进行的配置中你必须打开对 *kerneld* 的支持。需要时可以请求附加模块的能力对于使用堆叠式模块的驱动程序尤其有用。

隐藏在*kerneld*之后的思想很简单，但却很有效。当内核试图访问不可用资源时，它会通知用户程序而不仅仅是返回一个错误。如果守护进程成功地获得该资源，内核将继续工作；否则它将返回错误。实际上申请任何一种资源的时候都可以使用这种办法：诸如字符设备和块设备驱动程序，行律和网络协议等等。

用于获得按需装载能力的机制是使用一个修改过的消息队列，利用它在内核空间和用户空间之间相互传递文本信息。要让按需装载能正确工作，必须正确地配置用户级守护进程，并且内核代码必须做好准备，等待所需的模块。

可以从按需装载中受益的驱动程序的一个典型例子是通用帧捕获者(frame-grabber)驱动程序。它能支持几种不同的外设，但却表现出相同的外部行为。发布中将包括它所支持的所有设备卡的代码，但是在运行时只有正在被使用的那个特定设备卡的代码才真正需要。这样开发者就能够把具体实现划分为一个定义软件接口的通用模块和一系列用于低层操作的与硬件相关的模块。在通用模块检测到系统中安装的捕获者的类型后，它就能够为该捕获者申请正确的模块。

用户级方面

*kerneld*程序生存在用户空间，负责处理来自内核的对新模块的请求。它通过创建自己的消息队列和内核相连，然后进入睡眠，等待请求。

请求一个模块时，守护进程从内核中接收一个字符串并试图解析它。这个字符串可能是

下面两种形式之一：

- 目标文件的名字，就象*insmod*命令的典型参数一样。**floppy**是这种名字的一个例子；在这种情况下，守护进程将查找文件*floppy.o*并装载它。

- 更一般的标志符，比如**block-major-2**，它用来指明主设备号为2的块设备－也就是软盘驱动程序。这种类型的字符串是最常见的，因为内核通常只知道资源的数字标志符。例如，当你试图使用一个块设备时，内核只知道它的主设备号；仅仅就为了能通过名字来请求每个块设备而为它们实现各自不同的钩子函数很浪费。

显然，后一种情况时，必须有某种方法把模块的"id"映射成它的实际名字。这种关联并不由*kerneld*本身完成而是由*kerneld*调用*modprobe*来完成。在*depmod*命令的帮助下，由*modprobe*来处理模块装载的细节；*kerneld*本身只负责与内核的通讯并生成外部任务。所有这些程序都在*modules*包中一起发布。*depmod*是一个能产生类似Makefile那样的模块依赖信息的工具，而*modprobe*是能替代*insmod*用来正确装载模块堆栈的程序。例如*ppp*模块堆叠在*slhc*模块(Serial Line Header Compression)之上(换句话说，可以使用*slhc*模块中的符号)。除非已经装载了*slhc*，否则命令*insmod ppp*就会失败；另一方面，假如在安装好模块之后会调用命令*depmod -a*来创建依赖规则，命令*modprobe ppp*就能成功。

*insmod*和*modprobe*间的另一个差别是后者不会在当前目录中查找模块，它只在*/lib/modules*下的缺省目录中查找。这是因为该程序是一个系统实用例程，而不是一个交互工具；你可以通过在*/etc/modules.conf*中指定你自己的目录，来把它们加入缺省目录集。

/etc/modules.conf 是一个用于定制 *modules* 包的文本文件。它负责把象 **block-major-2** 这样的名字关联到 **floppy**。注意，2.0 前的版本的 *modules* 包查找的是另一个文件 */etc/conf.modules* ;出于兼容的考虑,仍支持这种文件名,但提倡更为标准的名字 *modules.conf*。

modules.conf 的语法在 *depmod* 和 *modprobe* 命令的 man 页中有很好的描述;然而，我觉得有必要在这里提及一些重要命令的意思。我用下面几行作为例子：

```
#sample line for /etc/modules.conf
keep
path[misc]=~rubini/driverBook/src/*
option short irq=1
alias eth0 ne
```

上面显示的第一行是注释；**path[misc]**指出在哪查找各种模块－而 **keep** 指出应把用户路径加到缺省路径中，而不是替换缺省路径。**Option** 制导(directive)指出在装载 short 模块时总是设定 **irq=1**，**alias** 行则指出当需要装载 **eth0** 时，相关的文件是 *ne.o*(ne2000 接口的驱动程序)。象 **alias block-major-2 floppy** 这样的行并不真正需要，因为 *modprobe* 已经知道的所有设备的官方主设备号，并且这些“可预见”的 **alias** 命令在程序中预定义过了。

那么，按需装载模块的正确安装，就是在文件*/etc/modules.conf* 中加入这么几行，因为 *kerneld* 是依靠 *modprobe* 来进行实际的装载操作。

内核级方面

请求加载模块和卸载模块，内核代码可以使用<**linux/kerneld.h**>中定义的函数。这些函数都定义成内联函数，实际上又将参数传递给了 *kerneld_send*。*kerneld_send* 函数是用来与 *kerneld* 通讯的一个灵活的引擎，它存在于文件 *ipc/msg.c* 中，如果你感兴趣的话，可以到那里浏览它。

这里，我不准备探讨 *kerneld_send* 的细节，因为在头文件<**linux/kerneld.h**>中定义的下列一些调用，足够你用来实现按需装载：

int request_module(const char *name)

需要加载模块的时候可以调用该函数。参数 **name** 或者是模块的文件名，或者是在用户空间解析的 id 类型字符串。在装载成功完成(或失败)后该函数返回。*request_module* 只能在进程上下文中被调用，因为当前进程将进入睡眠，等待模块被加载。任何一个按需加载的模块，在使用计数降为 0 时，都将自动卸载。

int release_module(const char *name，int waitflag)

请求立即卸载一个模块。如果 **waitflag** 不为 0，意味着函数在返回前必须等待卸载结束。如果 **waitflag** 为 0，函数可以在中断时间内调用－如果值得这么做的话。

int delayed_release_module(const char *name)

请求延迟的模块卸载。该函数总是立即返回。它的效果就是模块 **name** 在使用计数降为 0 就卸载，即使该模块并不是由 *kerneld* 加载的。

int cancel_release_module(const char *name)

该函数取消 *delayed_release_module* 的作用，它不阻止按需装载模块的自动卸载，最少当前的实现是这样的。一般不需要该函数，在这里提到它主要是出于完整性的考虑。

如果在内核空间检测到了错误，*kerneld_send* 的返回值，包括所有列出的这些函数的返回值都会是负的。如果内核中一切运行正常，返回值被置为执行这些操作的用户空间程序的退出值。成功时的退出值为 0，出错时为 1 到 255 间的一个数值。

有关 *kerneld_send* 的一个好消息就是即使在内核配置成不提供对 *kerneld* 的支持时，该函数仍然存在(并向模块开放)。因而，模块编写者总是可以调用上面显示的这些函数，但此时只返回-**ENOSYS**。当然，不能运行在 1.2 版的内核上，因为所有这些机制都是到 1.3.57 版才引入的。

现在，让我们实际地试着使用这些按需加载函数。为此目的，我们使用两个模块，分别叫作 *master* 和 *slave*，O'Reilly 的 FTP 站点上的 *misc-modules* 目录下以源文件的形式发布。我们还将使用 *slaveD.o* 来测试延迟卸载，并且使用 *slaveH.o* 来测试手工加载以及自动卸载模块。

为了不安装模块也可以运行测试代码，我在自己的*/etc/modules.conf* 文件中加入了如下

一些行：

```
keep
path[misc]=~rubini/driverBook/src/misc-modules
```

slave 模块只是一个空文件，而 *master* 模块看起来象下面这样：

```
#include <linux/kerneld.h>

int init_module(void)
{
int r[3];    /*  结果  */

r[0]=request_module("slave");
r[1]=request_module("slaveD");
r[2]=request_module("unexists");
printk("master: loading results are %i,%i,%i\n",r[0],r[1],r[2])；
return 0; /*  成功  */
}

void cleanup_module(void)
{
int r[4];/* results */

r[0]=release_module("slave",1/* wait */);
r[1]=release_module("slaveH",1 /* wait */);
r[2]=delayed_release_module("salveD");
r[3]=release_module("unexists",1 /* wait */);
printk("master: unloading results are %i,%i,%i,%i\n",r[0],r[1],r[2],r[3])；
}
```

在装载时，*master* 模块试着载入两个模块和一个并不存在的模块。除非你改变了终端的日志级别(loglevel)，否则 *printk* 消息将出现在终端上。下面是系统被配置成支持 *kerneld* 而该守护进程又是活动的时候，装载 *mater* 模块时的结果：

```
morgana.root# depmod -a
morgana.root# insmod master
master: loading results are 0,0,255
morgana.root# cat /proc/modules
slaveD          1          0 (autoclean)
slave           1          0 (autoclean)
master          1          0
isofs           5          1 (autoclean)
```

从 *request_module* 返回的值以及*/proc/modules* 文件（在第二章的“ 初始化和终止 ”一节中描述）均显示 slave 模块已经正确装载。另一方面，装载 unexists 的返回值 255 意味着用户程序失败，退出码是 255（或－1，因为它的长度为一个字节）。

我们简要的看看在卸载时会发生些什么，但在此之前先让我们手工加载 *slaveH*：

```
morgana.root# insmod slaveH
morgana.root# cat /proc/modules
slaveH          1               0
slaveD          1               0 (autoclean)
slave           1               0 (autoclean)
master          1               0
isofs           5               0 (autoclean)
morgana.root# rmmod master
master: unloading results are 0,0,0,255
morgana.root# cat /proc/modules
slaveD          1               0 (autoclean)
isofs           5               1 (autoclean)
morgana.root# sleep60;cat /proc/modules
isofs           5               1 (autoclean)
```

结果显示，除了 **unexists** 的卸载，一切都很正常，并且 *slaveD* 在一段时间过后也会被卸载。

尽管提供了各种例程，你还会发现，大部分时间 *request_module* 函数都能满足你的需要，而不要求你处理模块卸载；实际上，对不使用的模块，缺省地会自动进行卸载。绝大大部分时候，你甚至不必检查函数的返回值，因为只需要模块提供的一些函数。下面的实现比检查 *request_module* 的返回值更方便：

```
if ( (ptr = look_for_feature()) == NULL )     /* 是否没有该特性 */
request_module(modname)；               /* 试图装载它 */
if ( (ptr = look_for_feature()) == NULL )     /* 是否仍没有该特性 */
      return -ENODEV；                        /* 出错 */
```

模块中的版本控制

关于模块的一个主要问题是它们的版本相关性，在第二章的“ 版本相关性”一节中我曾经介绍过。针对每个要使用的版本的不同头文件都需要重新编译一次模块，当你运行好几个定制的模块时，这是件非常痛苦的事情。如果你运行的是以二进制形式发布的商业模块时，甚至连重新编译也是不可能的。

幸运的，内核开发者找到了一个变通的办法来处理版本问题。他们的想法是，只有改变了内核提供的软件接口，一个模块才不能兼容不同的内核版本。然后，软件接口可以由函数原型以及函数调用涉及到的所有数据结构的确切定义所表示。最后，可以使用一个CRC算法

把所有关于软件接口的信息映射到一个单一的 32 位数值[*]上去。

版本相关性的问题通过在每个由内核导出的符号的名字后面附加一个该符号相关信息的校验和来得到处理。解析头文件，就可以从中取出这些信息。这种便捷性是可选的，在编译的时候可以启动它们。

例如，当启动版本支持时，符号 **printk** 以类似 **printk_R12345678** 的形式向模块开放，这里 **12345678** 时函数使用的软件接口的检验和的 16 进制表示。加载一个模块到内核时，仅当加到内核中每个符号上的检验和匹配加到模块中相同符号上的校验和时，*insmod*(或 *modprobe*)才能够完成它的任务。

让我们来看看内核和模块都启动了版本支持的时候，会发生些什么:

- 内核本身并不修改符号。进程以通常的方式与内核链接，而且 *vmlinux* 文件的符号表看起来也和以前一样。
- 公共符号表使用版本名字创建，如*/proc/ksyms* 文件所示。
- 模块必须使用合并后的名字编译，这些名字在目标文件中是以未定义符号出现的。
- 装载程序用模块中未定义符号匹配内核中的公共符号，因此也要使用版本信息。

然而，上述情况只有当内核和模块都创建成支持版本化时才有效。如果有任何一方使用了原来的符号名，*insmod* 都会放弃版本信息，并试着用第二章的“ 版本相关性 ”一节中描述的方式来匹配模块声明的内核版本号和内核提供的版本号。

在模块中使用版本支持

当内核已经准备(可选的)输出版本化的符号时，模块源代码只需准备好支持该选项。可以在两处加入版本控制：在 *Makefile* 中或在源代码本身。因为 *modules* 包文档描述了在 *Makefile* 中如何做，我将向你显示在 C 源代码中如何做。用于演示 *kerneld* 如何工作的 *master* 模块能够支持版本化符号。如果用于编译模块的内核利用了版本支持的话，这种能力会自动启动。

用于合并符号名字的主要工具是头文件<linux/modversions.h>，它包括了所有公共内核符号的预处理定义。在包含这个头文件后，不管模块何时使用了内核符号，编译器都将看到合并了的版本。*modversions.h* 中的定义只有预先定义过 **MODVERSIONS** 才有效。

如果内核已经启动了版本支持，为了在模块中也启动它，我们必须保证在 **<linux/autoconf.h>**中已经定义过 **CONFIG_MODVERSIONS**。那个头文件控制着在当前内

[*] 实际上，CRC算法检测不到SMP和非SMP模块间的不兼容性，因为许多接口函数都是内联(**inline**)的，它们在SMP和非SMP机器上时不同编译的，即使它们对应了同样的检验和。你必须非常小心地避免混淆SMP模块和常规的模块。

核中(编译时)启动了哪些特性。每个 **CONFIG_**宏定义声明相应选项的状态是否要激活。

这样，*master.c* 的初始化部分包含如下部分：

```
#include <linux/autoconf.h> /*  检索 CONFIG_*宏  */
#if defined(CONFIG_MODVERSION) && !defined(MODVERSIONS)
#      define MODVERSIONS    /*  强迫打开它  */
#endif

#ifdef MODVERSIONS
#      include <linux/modversions.h>
#endif
```

在版本化的内核上编译这个文件时，目标文件的符号表会引用版本化符号，这些版本化符号匹配内核本身开放的那些符号。下面的屏幕快照显示了 *master.o* 中储存的符号名字。在 *nm* 的输出中，"T"代表“ 文本(text) ”，"D"代表“ 数据(data) ”，"U"代表“ 未定义(undefined) ”。最后一个标记表示目标文件引用了但没有声明的符号。

```
morgana% nm master.o
000000b0  T cleanup_module
00000000  T init_module
00000000  D kernel_version
          U kerneld_send_R7d428f45
          U printk_Rad1148ba
morgana% egrep 'printk|kerneld_send' /proc/ksyms
00131b40 kerneld_send_R7d428f45
0011234c printk_Rad1148ba
```

因为加到 *master.o* 中符号名上的校验和包含了与 *printk* 和 *kerneld_send* 相关的整个接口，模块与大部分内核版本都兼容。然而，如果与其中任一函数有关的数据结构被改变了，*insmod* 将因为模块与内核的不兼容而拒绝装载它。

开放版本化符号

以前的讨论中未涉及的情况是，当其它模块使用一个模块开放的符号时，会发生写什么。如果依赖版本信息来获得模块的可移植性，那么我们希望能把 CRC 校验码加到我们自己的符号上去。这个问题比仅仅链接到内核技巧性更高一些，因为我们需要将合并后的符号名向其它模块开放；我们需要一种办法来生成校验和。

分析头文件和生成校验和的任务是由随 *modules* 包一起发行的一个工具 *genksyms* 来做的。这个程序在自身的标准输入上接受 C 预处理器的输出，并在标准输出上打印出一个新的头文件。这个输出文件定义了原来那个源文件开放出来的每个符号的带检验和的版本。*genksyms* 的输出通常以后缀.*ver* 保存；下面我将遵循同样的惯例。

为了显示如何开放符号，我生成了两个名为 *export.c* 和 *import.c* 的虚构的模块。*export* 开放了一个名为 *export_function* 的简单函数，并且该函数会被第二个模块 *import.c* 使用。这个函数接收两个整数参数并返回它们的和－我们对这个函数并不感兴趣，而是对链接过程更感兴趣。

misc-modules 目录下的 *Makefile* 文件有从 *export.c* 生成 *export.ver* 文件的规则，因此 *export_function* 的检验和符号可以被 *import* 模块使用：

```
ifdef MODVERSIONS
export.o import.o: export.ver
endif

export.ver: export.c
	$(CC) -I$(INCLUDEDIR) -E -D__GENKSYMS__ $^|genksyms > $@
```

这几行演示了如何生成 *export.ver*，并且只有定义过了 **MODVERSIONS** 才会把它加到两个目标文件的依赖关系中去。如果内核启动了版本支持，还要添加几行到 *Makefile* 中负责定义 **MODVERSIONS**，但并不值得在这里展示它们。

然后，源文件必须为每个可能的预处理流程声明正确的预处理符号：不论是给 *genksyms* 的输入和真正编译过程，不论是启动还是关闭了版本支持。进一步，*export.c* 应当能够象 *master.c* 那样自动检测内核中的版本支持。下面几行向你显示了如何成功地做到这一点：

```
#ifndef EXPORT_SYMTAB
#	define EXPORT_SYMTAB	/* 需要这个定义是因为我们要开放符号*/
#endif

#include <linux/autoconf.h> /* 检索 CONFIG_* 宏 */
#if defined(CONFIG_MODVERSIONS)&& !defined(MODVERSIONS)
#	define MODVERSIONS
#endif

/*
 * 将内核符号和我们的符号的版本化定义包含进来，*除非*我们正在
 * 生成校验和(定义了*__GENKSYMS__)
 */
#if defined(MODVERSIONS) && !defined(__GENKSYMS__)
#	include <linux/modversions.h>
#	include "export.ver"	/* 为了包含 CRC，重定义了 "export_function" */
#endif
```

这些代码，虽然令人讨厌，但好处是：可以让 *Makefile* 处于一个干净的状态。另一方面，由 *make* 来传递正确的标志，涉及到为各种情况编写冗长的命令行，因此我就不在这里做了。

简单的 *import* 模块通过传递数字 2 和 2 作为参数，来调用 *export_function*；期望的结果是 4。下面的例子显示 *import* 确实链接到了 *export* 的版本化符号，并且调用了函数。版本化符号出现在*/proc/ksyms* 文件中。

```
morgana.root# insmod export
morgana.root# grep export /proc/ksyms
0202d024       export_function_R2eb14c1e   (export)
morgana.root# insmod import
import my mate tells that 2+2 = 4
morgana.root# cat /proc/modules
import                  1          0
export                  3     [import]         0
```

跨过卸载/装载的持久存储

一旦我们装备上了 *kerneld* 和版本支持，就会发现使用模块比使用链进内核的驱动程序更方便。模块化只有一个问题：如果一个驱动程序由 *kerneld* 载入，然后被配置（通过 *ioctl* 或者其它方法），那么下次将该驱动程序载入内核时又必须重新配置它。而启动时的配置信息则可以在*/etc/modules.conf* 文件中一劳永逸的指定，因此当要多次使用按需装载时，运行时的配置变得容易丧失。用户会可能会失望地发现刚离开休息一会设备的配置信息就已经丢失了。我们需要的是一种可以在模块卸载后持久地保存相关信息的技术。

实际上，*modules* 包从 2.0.0 版开始(*modules-2.2.0*)提供这种能力。

真正的代码还没有集成进官方的内核，但很可能会被 Linus 的源码所接受。目前，为了启动对持久存储的支持，你需要使用 *modules* 包中发布的一个补丁；这个补丁在**<linux/kerneld.h>**中添加了几行代码。

实际上，隐藏在模块信息的持久存储之后的想法很直接：与用户空间相互传输信息，内核代码可以与转载和卸载模块使用同一个 *kerneld* 引擎。然后，守护进程使用一个通用数据库来管理信息存储。

在用户空间而不是在内核空间实现持久存储的原因是为了简化代码。尽管可以设计出仅与内核空间有关的实现，从内核空间访问一个数据库文件需要将库代码在不可交换的内核空间中复制，而在用户空间，库代码的使用则没有任何开销。

Kerneld 中提议的实现使用了 *gdbm* 库来实现数据库。也可以选择使用盘上数据库。如果使用了该数据库，就可以获得跨过系统启动的持久性存储；如果没有使用该数据库，你只能在 *kerneld* 进程的生存期内获得持久性。

下面这些函数是在头文件**<linux/kerneld.h>**中定义的，用于获得持久存储特性：

```
int set_persist(char *key, void *value, size_t length);
int get_persist(char *key, void *value, size_t length);
```

这些函数的参数是一个文本关键字（**key**）和数据项本身－该关键字唯一地标记数据库中的一个数据项，而数据项则呈现为一个指针和长度的熟悉形式。参数 **key** 在整个系统内都必须唯一。这样就允许每个模块通过在关键字前加上模块名字将自己的关键字分离出来，但是它也允许不同的模块共享配置变量，如果因为什么原因需要这么做的话。

可能的返回值和其它调用 *kerneld_send* 的函数是一样的：0 表示成功，负数通知一个内核空间的错误，而正数用于通知一个用户空间的错误。通常可以忽略返回值，因为如果有错的话，*get_persist* 不会修改 **value** 的值，而如果 *set_persist* 不能保存这个值的话，也不会做任何事情。

新近的 *kerneld* 守护进程开始支持这种新特性，所以模块也可以选择在内核中不对 *kerneld.h* 做修补而将 *set_persist* 和 *get_persist* 的定义包含进来。但要注意向前兼容。建议使用在 *modules* 中发布的补丁；在被官方的内核源代码中包含前持久存储的内部实现可能会有变化。

我们已经看到，使用持久存储的主要原因是避免每次把模块载入一个运行内核时又要重新配置它。这对按需装载的模块的运行时配置尤其重要；这对装载时配置也是一个有意义的选项，因为更新*/etc/modules.conf* 对普通用户来说有些复杂。

持久存储另一个可能的用处是跟踪系统的硬件配置以避免不必要的探测。探测硬件是一种冒险的操作。它可能会错误地配置了其它的硬件，特别是对 ISA 设备，因为 ISA 不象 PCI 那样提供了一种通用的方法来扫描系统总线。（第 15 章“*外设总线概貌*”详细讨论了该问题。）

下面的例子代码显示了一个名为 *psm*（Persistent Storage Module）的假想模块是如何避免不必要的探测的。为简化讨论，这个例子程序最多支持一个设备。

```
int psm_base = 0; /* 基本的 I/O 端口，在装载时可以设定 */

int init_module(void)
{
    if (psm_base==0){ /* 在装载时没有设定 */
        get_persist("psm_base", &psm_base, sizeof(int));
        if (psm_check_hw(psm_base)!=0)
            psm_base=0; /* 旧的数值不再有效：探测 */
    }
    else
        if (psm_check_hw(psm_base)!=0)
            return -ENODEV; /* 没有任何地方指明基地址 */

    if (psm_base==0)
        psm_base=psm_probe(); /* 返回基本端口，或者，如果没能找到就返回 0 */
    if (psm_base==0)
```

```
        return -ENODEV;/* 没有找到任何设备 */

    set_persist("psm_base", &psm_base, sizeof(int));/* 找到：保存它 */
}
```

只有在装载时没有指定基本端口，并且以前的端口不再有效的时候，这些代码才探测硬件。如果找到一个设备，基本端口被保存起来，留作后用。

当驱动程序要支持多个设备时，检测新增加的硬件这个问题的一个可能的办法就是定义一个 **psm_newhw** 变量，如果添加了新设备到系统，那么用户可以在装载时对该变量进行设置。如果这样实现的话，那么当存在新设备时，用户必须使用 **insmod psm_newhw=1** 命令。如果 **psm_newhw** 不为 0，*init_module* 试着探测新设备，而在通常情况下它使用的是保存的信息。一个设备的基地址中的改变在上面给出的代码中已经处理过了，而不需要用户在装载时进行干预。

快速参考

本章引入下面一些内核符号：

/etc/modules.conf

modprobe 和 *depmod* 程序的配置文件。它用于配置按需加载，在这两个程序的 man 页中有描述。

#include <linux/kerneld.h>
int request_module(const char *name);
int release_module(const char *name, int waitflag);
int delayed_release_module(const char *name);
int cancel_release_module(const char *name);

这些函数通过 *kerneld* 守护进程进行模块的按需加载。

#include <linux/autoconf.h>
CONFIG_MODVERSIONS

只有当前内核被编译成支持版本化符号时这个宏才会被定义。

#ifdef MODVERSIONS
#include <linux/modversions.h>

这个头文件只有在 **CONFIG_MODVERSIONS** 有效时才存在，它包含了内核开放的所有符号的版本化名字。

EXPORT_SYMTAB

如果使用了版本支持并且你的模块使用了 *register_symtab* 来开放它自己的符号，必须定义这个宏。

__GENKSYMS__

当 *genksyms* 读入预处理文件来生成新的版本代码时，*make* 定义了这个宏。当生成新的检验和时，该宏用于有条件的防止包含<**linux/modversions.h**>头文件。

int get_persist(char *key, void *value, size_t length);

int get_persist(char *key, void *value, size_t length);

对模块数据的永久性存储的支持依赖于这两个函数，它们在头文件<**linux/kerneld.h**>中定义。

第十二章 加载快设备驱动程序

正如在第一章“Linux 核心简介”中“设备与模块的分类”中所概述的一样，Unix 的设备驱动程序并不仅限于字符设备。本章就来介绍一下第二大类的设备驱动程序——块设备驱动程序。所谓面向块的设备是指数据传输是以块为单位的（例如软盘和硬盘），这里硬件的块一般被称作“扇区（Sector）”。而名词“块”常用来指软件上的概念：驱动程序常常使用 1KB 大小的块，即使扇区大小为 512 字节。

在这一章，我们将来构造一个全特征的块设备驱动程序 *sbull*（Simple Block Utility for Loading Localities）。这个驱动程序与 *scull* 类似，也是使用计算机的内存作为硬件设备。换句话说，它是一个 RAM-disk 的驱动程序。*sbull* 可以在任何 Linux 计算机上执行（不过我只在有限的几个平台上作过测试）。

注册驱动程序

和字符设备驱动程序类似，核心里的块设备驱动程序也是由一个主设备号来标识。用来对其进行注册和取消注册的函数是：

```
int register_blkdev(unsigned int major, const char*name, struct file_operations *fops)
int unregister_blkdev(unsigned int major, const char*name)；
```

参数的含义与字符设备驱动程序一样，对主设备号的动态赋值也类似。因此，一个 *sbull* 设备与 *scull* 一样将自己注册：

```
result=register_blkdev(sbull_major,“sbull”，$sbull_fops);
if(result<0){
        printk(KERN_WARNING“sbull:can’t get major %d\n”,sbull_major);
        return result;
        }
if (sbull_major==0)    sbull_major=result;         /*dynamic*/
major=sbull_major;                                  /*Use “major”later on to save typing*/
```

register_blkdev 的 fops 参数与我们在字符设备驱动程序中使用的类似，为 *read*，*write* 以及 *fsync* 的操作并不要求针对某个驱动程序。通用函数 b*lock_read*, *block_write* 及 *block_fsync* 被用来代替任何针对某个驱动程序的函数。另外，*check_media_change* 和 *revalidate* 对块设备驱动程序也有意义，二者都在 sbull_fops 中定义。

在 *sbull* 中使用的 fops 结构如下：

（代码 236）

通用的读写操作被用来获得较高的性能。通过数据缓冲获得加速，这在字符设备驱动程序重中是没有的。块设备驱动程序可以被缓冲是因为它们的数据服从于计算机的文件层次结构，任何应用程序都无法直接访问，而字符设备驱动程序则不是这样。

不过，当缓冲的高速缓存不能满足一个读请求或当一个待处理的写操作要刷新到物理磁盘上时，驱动程序必须被调用来进行真正的数据传送。fops 结构除了 *read* 和 *write* 外，并不带有入口点，因此，必须要一个额外的结构 blk_dev_struct 来发出对实际数据传送的请求。

这个结构在<linux/blkdev.h>定义，它有几个域，但只有第一个域需被驱动程序设置。下面是这个结构在核心 2.0 中的定义。

（代码 237）

当核心需要为 *sbull* 设备产生一个 I/O 操作时，它便调用函数 blk_dev[*sbull*_major].request_fn。

因此这个模块的初始化函数须设置这个域使其指向它自己的请求函数。这个结构中的其它域只供核心函数或宏进行内部使用；你不必在你的代码段中显式地使用它们。
一个块设备驱动程序模块与核心的关系见图 12-1。
除了 blk_dev 还有几个数组带有块设备驱动程序的信息。这些数组一般由主设备号（有时也用次设备号）进行索引。它们在 *drivers/block/ll_rw_block.c* 中被声明和描述。
int blk_size[][];
这个数组由主设备号和次设备号索引。它以 KB 为单位描述了每个设备的大小。如果 blk_size[major]是 NULL，则不对这个设备的大小进行检查（也就是说，核心可能要求数据传送通过 end_of_device）。
int blksize_size[][];
被每个设备所使用的块的大小，以字节为单位。与上一个数组类似，这个二维数组也是由主设备号和次设备号索引。如果 blksize_size[major]是一个空指针，那么便假设其块大小为 BLOCK_SIZE（目前是 1KB）。块大小必须是 2 的幂，因为核心使用移位操作将偏移量转换为块号。
int hardsect_size[][];
与其它的一样，这个数据结构也是由主设备号和次设备号索引。硬件扇区的缺省大小为 512 字节。直到包括 2.0.X 版本为止，可变扇区大小仍未真正支持，因为一些核心代码仍旧假设扇区大小为半 KB。不过很可能在 2.2 版本中会真正实现可变扇区大小。
int read_ahead[];
这个数组由主设备号索引，它定义了一个文件被顺序读取时，核心可以提前读取多少扇区。在进程请求数据之前将其读出可以改善系统的性能及总的吞吐率。慢速设备最好指定一个较大的提前读的值，而一个快速设备则可以在较小的提前读的值下工作的很好。这个提前读的值越大缓冲高速缓存则需要越多的内存。每个主设备号有一个提前读的值，它对所有次设备号有效。这个值可以通过驱动程序的 ioctl 方法来改变；硬盘驱动程序一般设为 8 个扇区，对应着 4KB。
sbull 设备允许在加载时设置这些值，它们作用于示例驱动程序的所有次设备号。在 *sbull* 中变量名和它们的缺省值为：
size=2048(KB)
由 *sbull* 生成的每个 ramdisk 占两兆字节，正如系统的缺省值。
hardsect=512(B)
sbull 扇区大小是常用的半 KB 值。改变 hardsect 的值是不允许的。
如前所述，其它的扇区大小并不被支持。如果你一定要改它，可以将 *sbull*/*sbull*.c 中的安全检查去掉。不过请做好发生严重的内存崩溃的危险的准备。除非在你尝试时，已经加上了对可变扇区大小的支持。
rahead=2(扇区)
因为 ramdisk 是一个快速设备，所以这个缺省提前读的值比较小。
sbull 设备也允许你选择一个设备个数进行安装。devs 是设备个数，缺省设为 2，表明缺省内存使用量为 4 兆——2 个大小为 2MB 的盘。
sbull 设备的 *init_module* 的实现如下（不含主设备号的注册和错误恢复）：
（代码 239）
相应的清除函数如下所示：
（代码 240）
这里，调用 *fsync_dev* 是必须的，用以清除核心保存在不同高速缓存中的对设备的所有引用。事实上，*fsync_dev* 是运行在 *block_fsync* 之后的引擎，它是块设备的 *fsync* “ 方法 ”。

头文件 blk.h

由于块设备驱动程序的绝大部分是设备无关的,核心的开发者通过把大部分相同的代码放在一个头文件<linux/blk.h>中，来试图简化驱动程序的代码。因此，每个块设备驱动程序都必须包含这个头文件，在<linux/blk.h>中定义的最重要的函数是 *end_request*，它被声明为 static（静态）的。让它成为静态的，使得不同驱动程序可有一个正确定义的 *end_request*，而不需要每个都写自己的实现。

在 Linux1.2 中，这个头文件应该用<linux/../../drivers/block/blk.h>来包含。原因在于当时还不支持自定义的块设备驱动程序，而这个头文件最初位于 drivers/block 源码目录下。

实际上，*blk.h* 相当不寻常，比如它定义了几个基于符号 MAJOR_NR 的符号，而 MAJOR_NR 必须由驱动程序在它包含这个头文件之前声明。这里，我们再次看到 *blk.h* 在设计时并没有真正考虑自定义驱动程序。

看看 *blk.h*，你会发现几个设备相关的符号是按照 MAJOR_NR 的值声明的，也就是说 MAJOR_NR 应该提前知道。然而，如果主设备号是动态赋值的，驱动程序无法预知其值，因此也就不能正确定义 MAJOR_NR。如果 MAJOR_NR 未定义，*blk.h* 就不能设定一些在 *end_request* 中使用的宏。因此，为了让自定义驱动程序从通用的 *end_request* 函数受益，从而避免重新实现它，驱动程序必须在包含 *blk.h* 之前定义 MAJOR_NR 和其它几个符号。

下面的列表描述了一些必须提前定义的<linux/blk.h>中的符号。列表结尾给出了 *sbull* 中使用的代码。

MAJOR_NR

这个符号用来访问一些数组，特别是 blk_dev 和 blksize_size。自定义驱动程序（如 *sbull*）不能给这个符号赋一个常量值，可以将其定义（#define）为一个存有主设备号的变量。对 *sbull* 而言，它是 sbull_major。

DEVICE_NAME

被生成的设备名。这个字符串用来从 *end_request* 中打印错误信息。

DEVICE_NR(kdev_t device)

这个符号用来从 kdev_t 设备号中抽取物理设备的序号。这个宏的值可以是 MINOR(device)或别的表达式。这要依据给设备或分区分配次设备号的常规方式而定。对同一个物理设备上的所有分区，这个宏应返回同一个设备号——也就是说，DEVICE_NR 表达的是磁盘号，而不是分区号。这个符号被用来声明 CURRENT_DEV，它在 *request_fn* 中用来确定被一个传送请求访问的硬件设备的次设备号，可分区设备将在后面“可分区设备”一节中介绍。

DEVICE_INTR

这个符号用来声明一个指向当前下半部处理程序的指针变量。宏 SET_INTR(intr)和 CLEAR_INTR 用来给这个变量赋值。当设备可以发出具有不同含义的中断时，使用多个处理程序是很方便的。这个主题将在后面“中断驱动的块设备驱动程序”一节中讨论。

TIMEOUT_VALUE

DEBICE_TIMEOUT

TIMEOUT_VALUE 以记数的方式表达超时，这个超时的值与老计时器之一（特别地指计时器号 DEVICE_TIMEOUT）相关联。一个驱动程序可以在数据传送时间太长时，通过调用一个回调函数来检测错误条件。不过，由于老计时器由一个预赋值的计时器静态数组组成（见第六章“时间流”中“核心计时器”一节），一个自定义的驱动程序不能使用它们。我在 *sbull* 中对这两个符号都未定义，而是用一个新的计时器实现超时。

DEBICE_NO_RANDOM

在缺省情况下，函数 *end_request* 对系统熵值（即所有随机性的总量）有所贡献，这被 */dev/random* 所使用。如果一个设备不能对随机设备贡献显著的熵值，DEVICE_NO_RANDOM 应被定义。*/dev/random* 在第九章的“安装中断处理程序”中进行了介绍，SA_SAMPLE_RANDOM 也在那儿做了解释。

DEVICE_OFF(kdev_t device)

end_request 函数在结束时调用这个宏。例如在软盘驱动程序中，它调用一个函数，这个函数负责更新用来控制马达停转的一个计时器。如果设备没有被关掉，那么串 DEVICE_OFF 可以被定义为空。*sbull* 不使用 DEVICE_OFF。

DEVICE_ON(kdev_t device)

DEVICE_REQUEST

这些函数实际上并未在 Linux 的头文件中使用，所以驱动程序并不需要定义它们。大多数官方的 Linux 设备驱动程序声明这些符号并在内部使用它们，但我在 *sbull* 里并没有使用它。

sbull 驱动程序以如下的方式声明这些符号：

（代码 242）

头文件 *blk.h* 用上面列出的这些宏定义了一些可以由驱动程序使用的额外的宏，我将在后续章节里对之进行介绍。

处理请求

系统性能的方式排序。这些联结表中的请求被传递个驱动程序的请求函数，由它对链接表中的每个请求执行如下的任务：

- 检查当前请求的有效性。这个工作由在 *blk.h* 中定义的宏 INIT_REQUEST 完成。
- 进行实际的数据传送。用变量 CURRENT（实际上是个宏）可以获得发出请求的一些细节。CURRENT 是一个指向结构 request 的指针，我将在下节介绍这个结构的域。
- 清除当前的请求。这个操作由静态函数 *end_request* 完成，函数的代码在 *blk.h* 中。驱动程序向这个函数传递一个参数，即成功时为 1，失败时为 0。当 *end_request* 以参数 0 调用时，一个“I/O error”消息会被发给系统日志（通过 *printk*）。
- 循环回至开始，消化下一个请求。可以按照程序员的喜好使用一个 goto 或是一个 for(;;)，或者 while(1)。

实践中，请求函数的代码如下构造：

（代码 243）

尽管这段代码除了打印消息外什么都没有做，运行这个函数可以对数据传送的基本设计有一个很好的了解。到此为止，代码中唯一不清楚的地方是 CURRENT 的确切含义及它的域，这个我将在下一节介绍。

我的第一个 *sbull* 实现只包含了所示的空代码。我意在一个“不存在”的设备上构造一个文件系统，并使用它一会儿，只要数据仍在缓冲高速缓存中。在运行一个象这样罗嗦的请求函数时，看看系统日志能帮助你理解缓冲高速缓存是如何工作的。

在编译时，定义符号 *SBULL*_EMPTY_REQUEST，那么这个空且罗嗦的函数可以在 *sbull* 中运行。如果你想理解核心是如何处理不同块大小的，你可以在 *insmod* 命令行上实验 blksize=。这个空的请求函数通过打印每个请求的细节揭示了内部核心的工作情况。你或许也可以试试 hardsect=，但目前它被关闭了，因为比较危险。（见本章开始时的“注册驱动程序”）。

请求函数的代码并不显式地调用 return(),因为当列表中的待处理请求耗尽时，INIT-_REQUEST 会替你完成这个工作。

执行实际的数据传送

为了给 *sbull* 构造一个可以工作的数据传送，让我们先来看看核心是如何在结构 request 中描述一个请求的。这个结构在<linux/blkdev.h>中定义。通过访问 CURRENT 的域，驱动程序可以得到所有为在缓冲高速缓存的物理块设备之间传送数据所需要的信息。
CURRENT 是用来访问当前请求(即被首先服务的那个请求)。正如你可能猜到的，CURRENT 是 blk_dev[MAJOR_NR].current_request 的缩短形式。
下面这些当前请求的域包含了请求函数的有用信息：

kdev_t rq_dev;

请求所访问的设备。有本驱动程序所管理的所以设备均被使用同一个请求函数。一个请求函数处理所有的次设备号；rq_dev 可以被用来取得被操作的次设备。尽管 Linux1.2 称这个域为 dev，你仍然可以通过宏 CURRENT_DEV 来访问这个域。CURRENT_DEV 在我们所讨论的所有版本的核心中是可移植的。

int cmd;

这个域是 READ 或 WRITE。

unsigned long sector;

请求指向的第一个扇区。

unsigned long current_nr_sectors;

unsigned long nr_sectors;

当前请求的扇区数（大小)。驱动程序应该引用 current_nr_sectors,而应该忽略 nr_sectors（列在这里只是为了完整)。请看下一节“ 集簇请求 ”以获得更多的细节。

char *buffer

缓冲高速缓存中的域。如果 cmd==READ，就是写数据的位置；如果 cmd==WRITE，就是读数据的位置。

struct buffer_head *bh

这个结构描述了这个请求列表中的第一个缓冲区。我们将在“ 集簇请求 ”中用到这个域。在这个结构中还有其它的一些域，但它们基本上是核心内部使用的，驱动程序并不期望使用它们。

sbull 中可工作的请求函数的实现如下所示。在下面的代码中 sbull_devices 与 scull_device 类似。我们在第三章字符设备驱动程序的“ 打开方法 ”中介绍过 scull_devices。
（代码 245）

由于 *sbull* 只是个 RAM 盘，所以它的“ 数据传送 ”简化为一个 *memcpy* 调用。这个函数唯一“ 奇怪 ”的特征是条件语句中限制只能报告最多 5 个错误。这样做的目的是为了防止系统日志被太多的信息搞乱，因为 end_request(0)在请求失败时已打印了“ I/O error ”的消息。静态计数器是限制消息报告的标准做法，在核心中被多次用到。

集簇请求

上面请求函数中每次循环迭代都传送几个扇区——按照数据的使用，一般情况下，相当于一个块的“ 数据 ”量。例如，交换一次执行 PAGE_SIZE 大小的数据，而在 ext2 文件系统中就是传送 1KB 的块。
尽管在 I/O 中最方便的数据大小是一个块，但如果把相邻块的读或写集簇起来，你会获得很高的性能改善。在这个意义上，“ 相邻 ”指的是在硬盘上块的位置，而“ 连续 ”则指连续的内存区域。
将相邻块集簇有两个好处。首先，集簇加速了传送（例如，软盘驱动程序将相邻的块组合在一起，一次传送一个磁道的数据)。另外，它还能通过避免分配冗余的 request 结构来节省核心中的内存。
如果你愿意，也可以完全忽略集簇。上面给出的框架请求函数在没有集簇的情况下可以完全

正确地工作。不过，如果你想利用集簇，你需要更加仔细地研究 struct_request 的内部。
不幸的是，我所知道的所有的核心（至少到 2.1.51）都不能为自定义驱动程序进行集簇，而只对象 SCSI 和 IDE 这类内部驱动程序使用。如果你对核心的内部不感兴趣，你可以跳过本节的其余部分。不过，集簇将来还可能在模块中实现，它是通过减少相邻扇区的请求延迟来提高数据传送性能的一个有趣的途径。
在我描述驱动程序如何利用集簇请求之前，让我们先来看看当一个请求被排队时发生了什么。
当核心请求数据块传送时，它扫描目标设备的活动请求链表。当一个新块在盘上与一个已经被请求的块相邻时，它就被集簇到第一个块上。当前已存在的请求便被扩大了而不是增加一个新请求。
不幸的是，磁盘上相邻的两个数据缓冲区在内存中并不一定相邻。这个发现，外加上需要有效地管理缓冲高速缓存，导致创建一个 buffer_head 结构。一个 buffer_head 和一个数据缓冲相关联。
因此，一个“集簇”的请求，就是一个指向 buffer_head 的结构链表的 request_struct 结构。*end_request* 函数负责这个问题，这就是为什么前面给出的请求函数可以独立于集簇而工作。换句话说，*end_reques*t 要么清除当前请求并准备为下一个服务，要么准备处理同一个请求中的下一个缓冲区。因此，集簇对不关心它的设备驱动程序是透明的，上面的 *sbull* 函数就是一个例子。
一个驱动程序可能希望通过在它的 *request_fn* 函数中每次循环时处理整个缓冲区头链表的办法来从集簇中获益。为了做到这一点，驱动程序应该指向 CURRENT->current_nr_sectors（这个域我在上面的 *sbull_request* 中已经用过）和 CURRENT->nr_sectors，它包含了集簇在“当前”buffer_heads 列表中的相邻扇区的数目。
当前缓冲区头是 CURRENT->bh，而数据块是 CURRENT->bh->b_data。后一个指针为了象 *sbull* 一类忽略集簇的驱动程序缓冲在 CURRENT->buffer 中。
请求集簇在 *drivers/block/ll_rw_block.c* 的函数 *make_request* 中实现。不过，如上所说，集簇只对几个驱动程序有效（软驱，IDE，和 SCSI），以其主设备号为准。我曾通过以 major=34 装载 *sbull* 看到过集簇是如何工作的，因为 34 是 IDE3_MAJOR，而我的系统中没有第三个 IDE 控制器。
下面列表总结了当扫描一个集簇请求时应做的事项。bh 是被处理的缓冲区头——列表的第一项。对列表中的每个缓冲区头，驱动程序要完成下面一系列操作：

- 传送位于地址 bh->b_data，大小为 bh->b_size 字节数据块。数据传送的方向通常由 CURRENT->cmd 指出。
- 从列表中找出下一个缓冲区头：bh->b_request。接着通过将 b_request 置为 0，把刚传送过的缓冲区从列表中摘下。b_reqnext 指向你刚找出的新缓冲区。
- 通过调用 mark_buffer_uptodate(bh,1)，unlock_buffer(bh)，告诉核心你已完成对上个缓冲区的操作。这些调用保证缓冲高速缓存保持正确，不致有错误指向的指针。mark_buffer_uptodate 中参数“1”表示传送成功，若传送失败，则换为 0。
- 循环回到开始，传送下一个相邻块。

当你做完了集簇请求，CURRENT->bh 必须被更新以指向“已经被处理但未被解锁”的第一个缓冲区。如果列表中所有的缓冲区都已被处理和解锁，CURRENT->bh 可被置为 NULL。此时，驱动程序可以调用 *end_request*。如果 CURRENT->bh 是有效的，那么这个函数在转到下一个缓冲之前对其进行解锁——这是非集簇操作所发生的情况，此时由 *end_request* 照管所有的事情。如果指针为空，这个函数直接转到下一个请求。
全功能的集簇实现出现在 *driver/block/floppy.c*，而要求的所有操作出现在 *blk.h* 的 *end_request*

中。*floppy.c* 和 *blk.h* 都不容易理解，不过建议先从后者开始。

安装（Mounting）是如何工作的

块设备与字符设备及一般文件的不同在于它们可以被安装到计算机的文件系统上。这与一般的访问方式不同。一般的访问方式通过结构 file 进行，这个结构与特定的进程相关联，并且只在 *open* 到 *close* 之间存在。当一个文件系统被安装后，没有进程拥有一个 filp。
当核心把一个设备安装到文件系统上，它调用一般的 *open* 方法来访问驱动程序。然而，这种情况下 *open* 的参数 filp 是个虚的变量，几乎只是为了占个地方，它唯一有意义的域是 f_mode。其它域含任意值并不使用。f_mode 的值是告诉驱动程序设备是以只读（f_mode==FMODE_READ）还是读写（f_mode==（FMODE_READ|FMOD_WRITE））方式被安装。使用一个虚变量而不是 file 结构的原因是因为实际的结构 file 在进程结束时将被释放，而被安装的文件系统在 *mount* 命令完成后仍然存在。

在安装时，驱动程序唯一调用的是 *open* 方法。当磁盘被安装后，核心调用设备中的 *read* 和 *write* 方法（被映射到 request_fn）来管理文件系统中的文件。驱动程序并不知道 request_fn 服务的是一个进程（象 *fsck*）还是核心中的文件系统层。
至于 umount，它只是刷新缓冲高速缓存并调用驱动程序的 *release*（*close*）方法。由于没有有意义的 filp 可以传递给 fop->realse，核心使用 NULL。
因此，当你实现 *release* 时，你应将驱动程序设为能处理为 NULL 的 filp 指针。不然，如果你用了 filp，你可能运行 *mkfs* 和 *fsck*，它们都使用 filp 来访问设备，你也可能 mount 这个设备，但 umount 将无法运行，原因就是 NULL 指针。
由于一个块设备驱动程序的 release 实现不能用 filp->private_data 来访问设备信息，它采用 inode->i_rdev 来区分设备。这里是 *release* 的 *sbull* 实现：
（代码 249）
其它的驱动程序函数并不关心 filp 问题，因为它们与安装的文件系统无关。例如，一个显示地 *open* 这个设备的进程只发出 *ioctl*。

ioctl 方法

如字符设备一样，块设备也可以通过 *ioctl* 系统调用进行操作。两者之间相对不一样的地方在于块设备驱动程序有大量驱动程序都要支持的 *ioctl* 命令。
块设备驱动程序经常要处理的命令如下所示，它们在<linux/fs.h>中被声明。

BLKGETSIZE
: 获取当前设备的大小，以扇区数表示。由系统调用传递的 数值 arg 是一个指向 long 数值的指针，用来将大小拷贝到一个用户空间的变量中。这个 *ioctl* 命令可以被 *mkfs* 用来获知产生的文件系统的大小。

BLKFLSBUF
: 字面上的意思是“刷新缓冲区”。这个命令的实现对每个设备都是一样的，我们将在后面整个 *ioctl* 方法的示例代码中给出来。

BLKRAGET
: 用来为设备取得当前提前读的值。当前数值应该用在参数 arg 中传递给 *ioctl* 的指针写进一个 long 类型的用户空间变量。

BLKRASET
: 设置提前读的值。用户进程在 arg 中传递这个新值。

BLKRRPART
: 重读分区表。这个命令只对可分区设备有意义，将在后面“可分区设备”中介绍。

BLKROSET
BLKROGET
这些命令用来改变和检查设备的只读标志。因为代码是设备无关的，它们由宏RO_IOCTLS（kdev_tdev,unsigned long where）来实现。这个宏在 *blk.h* 中定义。
HDIO_GETGEO
在<linux/hdreg.h>中定义，用来获得磁盘的几何参数。这个参数应被写入用户空间的结构 hd_geometry 中，它也在 *hdreg.h* 中定义。*sbull* 显示了这个命令的一般实现。
HDIO_GETGEO 是<linux/hdreg.h>中定义的一系列 HDIO 命令中最常用的一个。感兴趣的读者可以查看 *ide.c* 和 *hd.c* 以获得这些命令的更多信息。
这里列出的这些命令的一个主要缺点是它们是以“老”方法定义的（是第五章“增强的字符设备驱动程序操作”中“选择 *ioctl* 命令”一节），因此无法使用位域的宏来减化代码——每个命令要实现它自己的 *verify_area*。不过，如果一个驱动程序需要定义它自己的命令来利用设备的一些特殊特点，你可以自由地使用“新”方法来定义命令。
sbull 设备只支持上面的通用命令，因为实现设备特定的命令与实现字符设备驱动程序的命令没有什么不同。*sbull* 的 *ioctl* 实现如下所示，它将有助于你理解上面列出的命令。
（代码 250）
（代码 251）
函数开始的 PDEBUG 语句被留出，这样当你编译这个模块时，你可以打开调试（debugging）来看看设备上调用了哪个 *ioctl* 命令。
例如，对于显示的 *ioctl* 命令，你可以在 *sbull* 上使用 *fdisk*。下面是在我自己系统上的一个示例执行过程：
（代码 252 1#）
在会话过程中下面的消息出现在我的系统日志中：
（代码 252 2#）
第一个 ioctl 是 HDIO_GETGEO，它在 *fdisk* 启动时被调用；第二个是 BLKRRPART。对后一个命令的 *sbull* 实现仅仅是调用一下 revalidate 函数，它则在打印输出中打印最后的消息（见本章后面的“revalidate”）。

可拆卸的设备

在我们讨论字符设备驱动程序时，我们忽略了 fops 结构中的最后两个文件操作，因为它们只是为可拆卸块设备而设的。现在是看看它们的时候了。*sbull* 并不真是可拆卸的，但它假装是，因此它实现了这些方法。
我所说的操作是 *check_media_change* 和 *revalidate*。前者用来发现设备自上次访问以来是否改变过，后者则在磁盘变动之后重新初始化驱动程序的状态。
至于 *sbull*，与设备相联的数据区在使用计数下降为零后半分钟要释放。待这个设备处于未安装状态（或关闭状态）足够长的时间以模拟一次磁盘的改变，下一次对设备的访问分配一个新的内存区域。
这一类的“时间到期”通过一个核心计数器来实现。

check_media_change

这个检查函数接收到 kev_t 做为一个确定设备的参数。如果介质被改变了返回值为 1，否则为 0。如果一块设备驱动程序不支持可拆卸设备，可以通过置 fops->check_media_change 为 NULL 来避免这个声明函数。
有趣的是要注意，当一个设备是可拆卸的，但却无法判断它是否改变了，这时，返回 1 是个安全选择。事实上，IDE 驱动程序在处理可拆卸磁盘时就是这么做的。

sbull 的实现是这样的，当由于计数器超时，设备已经从内存中删除时就返回 1，如果数据仍然有效则返回 0。如果设置了调试，它同时向系统日志打印一条消息，这样用户就可以检查核心什么时候调用了这个方法。

（代码 253 1#）

revalidate

这个有效化函数是在检测到一个磁盘的改变时被调用。它也被在核心的 2.1 版中实现的各种 *stat* 系统调用。返回值目前不做使用；为安全起见，返回 0 表示成功，出错时返回一个负的错误代码。

revalidate 执行的动作是设备特定的，但 revalidate 通常更新一些内部状态信息以反映新的设备。

在 *sbull* 中，*revalidate* 方法在没有一个有效区域的情况下试图分配一块新的数据区域。

（代码 253　2#）

（代码 254　1#）

特别注意

当可拆卸设备已经打开时，驱动程序也应该检查是否有磁盘的改变；在 *mount* 时核心自动调用它的 *check_disk_change* 函数，但在 *open* 时，并不这样做。

不过，有些程序直接访问磁盘数据而不安装这个设备，*fsck*，*mcopy* 和 *fdisk* 都是这类程序的例子。如果驱动程序在内存中保存可拆卸设备的状态信息，它应在设备第一次打开时调用 *check_disk_change* 函数。这个核心函数还要依赖驱动程序方法（*check_media_change* 和 *revalidate*），因此在 *open* 里不须实现任何特别的东西。

这里是 *open* 的 *sbull* 实现，它关注了发生磁盘改变的情况：

（代码 254　2#）

在驱动程序中不需对磁盘的改变做任何别的。如果一个磁盘被改变了，而它的打开计数大于零，那么数据会被破坏。防止这种情况发生的唯一方法是让利用在物理上支持的设备使用计数控制门锁。*open* 和 *close* 可以在合适的时候关闭或打开锁。

可分区设备

如果你想用 *fdisk* 生成分区，你会发现它们有一些问题。*fdisk* 程序称这些分区为*/dev/sbull01*，*/dev/sbull02* 以 此类推，但文件系统上并不存在这些名字。的确，基本的 *sbull* 设备是一个字节阵列，不存在提供访问数据区域的子区域的入口点，因此想对 *sbull* 进行分区是行不通的。

为了能对设备分区，我们必须给每个物理设备分配几个次设备号。一个数字用来访问整个设备（如*/dev/hda*），其它的用来访问不同的分区（如*/dev/hda1*）。由于 *fdisk* 产生分区名的办法是在全盘设备名后加一个数字后缀，我们将在后面的块设备驱动程序中遵循同样的命名规则。

在本节中我将要介绍的设备叫 *spull*，因此它是一个“ 简单的可分区工具（Simple Partitionable Utility）”。这个设备位于 *spull* 目录，完全与 *sbull* 无关，尽管它们共享很多代码。

在字符设备驱动程序 *scull* 中，不同的次设备号可以实现不同的行为，因此一个驱动程序可以显示几种不同的实现。而按照次设备号区分块设备是不可行的，这就是为什么 *sbull* 和 *spull* 被分离开。这种无能为力是块设备驱动程序的一个基本特征，因为几个数据结构和宏只是作为主设备号的函数定义的。

关于移植，需要注意的是可分区模块不能被加载到核心的 1.2 版，因为符号 *resetup_one_dev*（在本节后面介绍）没有被引出到模块。在对 SCSI 盘的支持模块化之前，没有人会考虑可分区的模块。

我要介绍的设备结点被称做 pd，表示“可分区磁盘（partitionable disk）”。四个完整的设备（又称“单元”）被称做*/dev/pd*a 直到*/dev/pdd*；每个设备最多支持 15 个分区。次设备号有下面的含义：低四位表示分区号（0 为完整的设备），高四位表示单元号。这个规则在源文件中由下面的宏表达：

（代码 255）

普通硬盘

每个可分区设备需要知道它是如何分区的。这个信息可以从分区表中得到。初始化进程的一部分包括解码分区表，并更新内部数据结构以反映分区信息。

这个解码并不容易。不过幸运的是，核心提供可被所有块设备驱动程序使用的“普通硬盘”支持，它显著地减少了处理分区驱动程序的代码。这个普通支持的另一个好处是驱动程序的作者不必理解分区是如何完成的，而不需要修改驱动程序的代码就可以在核心中支持新的分区方式。

想要支持分区的块设备驱动程序要包含<linux/genhd.h>，并声明结构 gendisk。所有这样的结构被组织在一个链表中，它的头是全局指针 gendisk_head。

在我们进行下一步之前，让我们先看看结构 gendisk 的域。你为了利用普通设备支持就需要理解它们。

int major

确定这个结构所指的设备驱动程序的主设备号。

const char*major_name

属于这个主设备号的设备的基本名。每个设备名是通过在这个名字后为每个单元加一个字母并为每个分区加一个数字得到。例如，“hd”是用来构成*/dev/hda1* 和*/dev/hda3* 的基本名。基本名最多 5 个字符长，因为 *add_partition* 在一个 8 字节的缓冲区中构造全名，它要附加上一个确定单元的字母，分区号和一个终止符‘\0’。*spull* 所用的名字是 pd（“可分区磁盘（partitionable disk）”）。

int minor_shift

从设备的次设备号中获取驱动器号要进行移位的次数。在 *spull* 中这个数是 4。这个域中的值应与宏 DEVICE_NR(device)中的定义一致（见本章前面的“头文件 *blk.h*”）。*spull* 中的宏扩展为 device>>4。

int max_p

分区的最大数目。在我们的例子中，max_p1 是 16，或更一般地，是<<minor_shift。

int max_nr

单元的最大数目。在 *spull* 中，这个数字是 4。单元最大数目在移位 minor_shift 次后的结果应匹配次设备号的可能的范围，目前是 0-255。IDE 驱动程序可以同时支持很多驱动器和每一个驱动器很多分区，因为它注册了几个主设备号，从而绕过了次设备号范围小的问题。

void(*init)（struct gendisk*）

驱动程序的初始化函数，它在初始化设备后和分区检查执行前被调用。我将在下面介绍这个函数更多的细节。

struct hd_struct *part

设备的解码后的分区表。驱动程序用这一项确定通过每个次设备号哪些范围的磁盘扇区是可以访问的。大多数驱动程序实现 max_nr<<minor_shift 个结构的静态数值，并负责数组的分配和释放。在核心解码分区表之前驱动程序应将数组初始化为零。

int *sizes

这个域指向一个整数数组。这个数组保持着与 blk_size 同样的信息。驱动程序负责分配

和释放该数据区域。注意设备的分区检查把这个指针拷贝到 blk_size，因此处理可分区设备的驱动程序不必分配这后一个数组。

int nr_real

存在的真实设备（单元）的个数。这个数字必须小于等于 max_nr。

void *real_devices

这个指针被那些需要保存一些额外私有信息的驱动程序内部使用（这与 filp->private_data 类似）。

void struct gendisk *next

在普通硬盘列表中的一根链。

分区检查的设计最适合那些直接链入核心映象的驱动程序，因此我将从介绍核心代码的基本结构开始。以后我将介绍 *spull* 模块处理它的分区的方法。

核心中的分区检测

在引导时，*init/main.c* 调用了各种各样的初始化函数。其中一个是 *start_kernel*，它通过调用 *device_setup* 来初始化所有的驱动程序。这个函数又调用 *blk_dev_init*，接着检查所有注册的普通硬盘的分区信息。任何一个块设备驱动程序，如果它找到至少一个它的设备，就将这个驱动程序的 genhd 结构注册到核心列表中，这样它的分区便可以被正确地检测出来。

因此，一个可分区的驱动程序应该声明它自己的结构 genhd。这个结构看起来如下：

（代码 258）

于是，在这个驱动程序的初始化函数中，这个结构被排队在可分区设备的主列表中。

被链入核心的驱动程序的初始化函数与 *init_module* 等价，即使它被调用的方式不同。这个函数一定包含如下两行，它们用来将结构排队：

```
my_gendisk.next=gendisk_head;
gendisk_head=my_gendisk;
```

通过将结构插入链表，这简单的两行便是驱动程序入口点为所有的分区正确地识别和配置所需要的所有内容。

额外的设置通过 *my_geninit* 完成。在上面的例子中，这个函数填充“单元数”域来反映计算机系统的实际硬件设置。在 *my_geninit* 结束后，*gendisk.c* 为所有的盘（单元）执行实际的分区检测。你可以看到系统启动时被检测的分区，因为 *gendisk.c* 在系统控制台上打印分区检查 Partition check：，后面跟随它在可得的普通硬盘上找到的所有分区。

你可以修改前面的代码，推迟 my_sizes 和 my_partitions 的分配直到 *my_geninit* 函数。这可以节省少量的核心内存，因为这些数组可以小到 nr_real<<minor_shift，而竟态数组则必须为 max_nr<<minor_shift 字节长。不过，典型的数值是每个物理单元节省几百个字节。

模块中的分区检测

一个模块化的驱动程序和链接到核心的驱动程序的区别在于它无法受益于集体中的初始化。相反，它需要处理它自己的设置。由于没有为模块的两步初始化，所以 spull 的 gendisk 结构在它的 init 函数指针中有一个 NULL 指针：

（代码 259 1#）

同时也不必在普通硬盘的全局链表里注册 gendisk 结构。

通过引出函数 *resetup_one_dev*，文件 *gendisk.c* 被准备用来处理象模块需要一类“晚的”初始化。*resetup_one_dev* 为单个物理设备扫描分区。其原型是：

boid resetup_one_dev(struct gendisk *dev,int drive)；

从这个函数名字你可以看出来它是要改变一个设备的设置信息。这个函数被设计为由 *ioctl* 里 BLKRRPART 实现调用，但他也可以被用来完成一个模块的初始设置。

当一个模块被初始化后，它应该为每个它将要访问的物理设备调用 *resetup_one_dev*，从而将

分区信息贮存 my_gendisk->part 中。分区信息会被设备的 *request_fn* 函数使用。
在 *spull* 中，*init_module* 函数除了通常的指令外还包含了下面的代码。它分配分区检测所需的数组并初始化数组中完整磁盘的项目。
（代码 259 2#）
（代码 260 1#）
有趣的是注意到 *resetup_one_dev* 通过重复调用下面函数打印分区信息：
printk（" %s："，disk_name（hd，minor，buf））；
这就是为什么 *spull* 要打印一个引导串。它意味着要为塞进系统日志的信息增加一些上下文。当一个可分区的模块被卸载时，驱动程序应该通过为每个支持的主/次对调用 *fsync_dev* 来安排所有的分区刷新。而且，如果结构 gendisk 被插在全局链表中，它应该被删除——注意 *spull* 并未自己插入它，原因上面提到过。
spull 的清除函数是：
（代码 260 2#）
（代码 261）

使用 *Initrd* 进行分区检测

如果你想从一个设备上安装你的根文件系统，而这个设备的驱动程序只有模块化的形式，你就必须使用由现代 Linux 核心提供的 *Initrd* 工具。我不想在这里介绍 *Initrd*，这一小节是针对那些了解 *Initrd* 并想知道它是如何影响块设备驱动程序的读者的。
当你用 *Initrd* 引导一个核心时，它会在安装真正的根文件系统之前建立一个暂时的运行环境。模块通常是从被用作临时根文件系统的 ramdisk 中装载。
由于 *Initrd* 进程是在所有其它引导时初始化完成之后才开始运行（但在真正的根文件系统安装之前），因此在装载一个普通模块和在 *Initrd* ramdisk 中的模块没有区别。如果一个驱动程序可以正确地装载并以模块的形式被使用，那么所有含有 *Initrd* 的 Linux 发布都可以将这个驱动程序包含在安装盘中而不需要你研究修改核心源码。

spull 的设备方法

除了初始化和清除工作，可分区设备和不可分区设备还有其它的不同。根本上说，这些区别来自一个事实，即如果一个磁盘是可分区的，那么同一个物理设备可以通过不同的次设备号进行访问。从次设备号到磁盘上物理位置的映射由 *resetup_one_dev* 存贮在数组 gendisk->part 中。下面的代码只包含了 *spull* 与 *sbull* 不同的部分，因为绝大部分代码是完全一样的。
首先，*open* 和 *close* 必须掌握每个设备的使用记数情况。由于使用记数是关于物理设备（单元）的，下面的赋值被用于 dev 变量：
spull_Dev *dev=spull_devicex+DEVICE_NR(inode->i_rdev)；
这里用到的宏 DEVICE_NR 必须包含<linus/blk.h>之前被声明。
尽管几乎所有的物理方法可以工作于物理设备，*ioctl* 应该访问每个分区的特定信息。例如，应该告诉 *mkfs* 每个分区的大小，而不是完整设备的大小。下面是 *iotcl* 的 BLKGETSIZE 命令如何因一个设备一个次设备号变为一个设备多个次设备号而受到影响的。正如你所期望的，spull=gendisk->part 被用来做为分区大小的来源。
（代码 262　#1）
另一个对可分区设备不同的 ioctl 命令是 BLKRRPART。对可分区设备来说重读分区表是有意义的，它等价于在发生磁盘改变后对磁盘的重有效化（revalidating）：
（代码 262　#2）
函数 *spull_revalidate* 接着调用 *resetup_one_dev* 来重建分区表。不过，它首先需要清掉所有以前的信息——不然的话，尾分区还会出现在分区表的后部，如果新分区表含有少于以前的分区数。

（代码 262　#3）
但是 *sbull* 和 *spull* 的主要区别在于请求函数。在 *spull* 中，请求函数必须使用分区信息以在不同次设备号之间正确地传送数据。
spull_gendisk->part 中的信息在物理设备上为每个分区定位。part[minor]->nr_sects 是分区的大小，part[minor]->start_sect 是距磁盘起始位置的偏移。请求函数最终转回到完整磁盘的实现。
下面是在 *spull_request* 中的相关行：
（代码 263#4）
（代码 263#1）
扇区数乘以扇区大小 512（这里直接编写在 *spull* 中的）得到以字节为单位的分区的大小。

中断驱动的块设备驱动程序

当一个驱动程序控制一个实际的硬件设备时，操作一般是中断驱动的。使用中断依靠在 I/O 操作时释放处理器，从而提高系统性能。为了让中断驱动 I/O 能够工作，被控制的设备必须能异步地传送数据并产生中断。
当驱动程序是中断驱动时，请求函数派生出一个数据传送并立即返回，并不调用 *end_request*。不过，核心并不认为请求已经被完成了，直到 *end_request* 被调用时。因此，当设备发出信号表明数据传送已经完成时，上半部或下半部中断处理程序调用 *end_request*。
若不使用系统微处理器，*sbull* 和 *spull* 都不能传送数据；不过 *spull* 装备了伪装中断驱动操作的能力，这通过在装载时指定 irq=1 实现。当 irq 为零时，驱动程序使用一个核心计时器来推迟当前请求的完成。延迟的长度是 irq 的值：值越大，延迟越长。
中断驱动设备的请求函数告诉硬件执行传送并返回。*spull* 函数执行通常的错误检查，并调用 *memcpy* 传送数据（这个任务在实际驱动程序中是异步执行的）。它将确认应答延迟至中断时。
（代码 263　#2）
（代码 264　#1）
当设备正在处理当前请求时，新来的请求可以积累起来，但如果驱动程序正在处理一个请求，核心并不为之调用请求函数。这就是为什么这里显示的函数并不检查双重调用。
在上一个数据传送完成后，设置下一次是中断处理程序的责任。为避免代码重复，处理程序通常调用请求函数，因此请求函数应能在中断时运行（见第 6 章“ 任务队列的本质 ”）。
在我们的示例模块中，中断处理程序的角色是通过计时器超时时调用的函数完成的。那个函数调用 *end_request* 并通过调用请求函数来调度下一次数据传送。
（代码 264 2#）
注意这个中断处理程序调用请求函数来调度下一次操作。这意味着在这种情况下，请求函数必须能在中断时进行。
如果你试图以中断驱动风格运行 *spull* 模块，你会明显地注意到增加的延迟。这个设备几乎象它以前一样快，因为缓冲高速缓存避免了内存和物理设备之间的绝大多数数据传送。如果你想感受一下一个慢设备是如何运行的，你可以在加载 *spull* 时为 irq 指定一个大点儿的值。

快速参考

下面总结的是在写一个块设备驱动程序时最重要的函数和宏。不过，为了节省空间，我没有列出结构 request 和 genhd 的域，另外我也省略了预先定义的 *ioctl* 命令。
int register_blkdev(unsigned int major,const char *name,struct file_operations *fops);
int unregister_blkdev(unsigned int major,const char *name);

这两个函数负责在 *init_module* 中的设备注册和 *cleanup_module* 中的设备撤除。

struct blk_dev_struct blk_dev[MAX_BLKDEV];

这个数组用做在核心和驱动程序间请求的传递。blk_dev[major].request_fn 应该在加载时被赋值以指向“当前请求的请求函数”。

int read_ahead[][];

每个主设备号的提前读的值。数值为 8 对硬盘一类的设备比较合理；对慢速介质这个值应该大一点儿。

int read_ahead[];

int blksize_size[][];

int blk_size[][];

这些二维数组由主设备号和次设备号索引。驱动程序负责分配和释放与主设备号关联的矩阵中的行。这些数组分别表示以字节为单位的设备块的大小（通常为 1KB），以 KB 为单位的每个次设备的大小（并非以块为单位），以字节为单位的硬件扇区的大小。

当前，不支持 512 以外的扇区大小，尽管代码中有一个钩子函数（hook）。

MAJOR_NR

DEVICE_NAME

DEVICE_NR(kdev_t device)

DEVICE_INTR

#include <linux/blk.h>

这些宏必须在驱动程序包含头文件之前定义，因为绝大多数头文件要用到它们。

struct request *CURRENT

这个宏指向当前的请求。这个请求结构描述一个要被传送的数据块，被当前驱动程序的 *request_fn* 使用。

#include <linux/gendisk.h>

struct genhd;

普通硬盘使得 Linux 轻松地支持可分区设备。

void resetup_one_dev(struct gendisk *genhd,int ddrive);

这个函数扫描磁盘的分区表，并重写 genhd->part 以反映新的分区。

第十三章 MMAP 和 DMA

这一章介绍 Linux 内存管理和内存映射的奥秘。同时讲述设备驱动程序是如何使用“ 直接内存访问 ”（DMA）的。尽管你可能反对，认为 DMA 更属于硬件处理而不是软件接口，但我觉得与硬件控制比起来，它与内存管理更相关。
这一章比较高级；大多数驱动程序的作者并不需要太深入到系统内部。不过理解内存如何工作可以帮助你在设计驱动程序时有效地利用系统的能力。

Linux 中的内存管理

这一节不是描述操作系统中内存管理的理论，而是关注于这个理论在 Linux 实现中的主要特征。本节主要提供一些信息，跳过它不会影响您理解后面一些更面向实现的主题。

页表

当一个程序查一个虚地址时，处理器将地址分成一些位域（bit field）。每个位域被用来索引一个称做页表的数组，以获得要么下一个表的地址，要么是存有这个虚地址的物理页的地址。为了进行虚地址到物理地址的映射，Linux 核心管理三级页表。开始这也许会显得有些奇怪。正如大多数 PC 程序员所知道的，x86 硬件只实现了两级页表。事实上，大多数 Linux 支持的 32 位处理器实现两级，但不管怎样核心实现了三级。
在处理器无关的实现中使用三级，使得 Linux 可以同时支持两级和三级（如 Alpha）的处理器，而不必用大量的#ifdef 语句把代码搅得一团糟。这种“ 保守编码 ”方式并不会给核心在两级处理器上运行时带来额外的开销，因为实际上，编译器已经把没用的一级优化掉了。
但是让我们看一会儿实现换页的数据结构。为了跟上讨论，你应该记住大多数用作内存管理的数据都采用 unsigned long 的内部表示，因为它们所表示的地址不会再被复引用。
下述几条总结了 Linux 的三级实现，由图 13-1 示意：

- 一个“ 页目录（Page Directory,PGD）”是顶级页表。PGD 是由 pgd_t 项所组成的数组，每一项指向一个二级页表。每个进程都有它自己的页目录，你可以认为页目录是个页对齐的 pgd_t 数组。
- 二级表被称做“ 中级页目录（Page Mid_level Directory）”或 PMD。 PMD 是一个页对齐的 pmd_t 数组。每个 pmd_t 是个指向三级页表的指针。两级的处理器，如 x86 和 sparc_4c，没有物理 PMD；它们将 PMD 声明为只有一个元素的数组，这个元素的值就是 PMD 本身——马上我们将会看到 C 语言是如何处理这种情况以及编译器是如何把这一级优化掉的。
- 再下一级被简单地称为“ 页表（Page Table）”。同样地，它也是一个页对齐的数组，每一项被称为“ 页表项（Page Table Entry）”。核心使用 pte_t 类型表示每一项。pte_t 包含数据页的物理地址。

上面提到的类型都在<asm/page.h>中定义，每个与换页相关的源文件都必须包含它。
核心在一般程序执行时并不需要为页表查寻操心，因为这是有硬件完成的。不过，核心必须将事情组织好，硬件才能正常工作。它必须构造页表，并在处理器报告一个页面错时（即当处理器需要的虚地址不在内存中时）查找页表，。
下面的符号被用来访问页表。<asm/page.h>和<asm/pgtable.h>必须被包含以使它们可以被访问。
（Figure 13.1 Linux 的三级页表）
PTRS_PER_PGD
PTRS_PER_PMD

PTRS_PER_PTE

每个页表的大小。两级处理器置 PTRS_PER_PMD 为 1，以避免处理中级。

unsigned long pgd_bal(pgd_t pgd)

unsigned long pmd_val(pmd_t pmd)

unsigned long pte_val(pte_t pte)

这三个宏被用来从有类型数据项中获取无符号长整数值。这些宏通过在源码中使用严格的数据类型有助于减小计算开销。

pgd_t *pgd_offset(struct mm_struct *mm,unsigned long address)

pmd_t *pmd_offset(pgd_t *dir,unsigned long address)

pte_t *pte_offset(pmd_t *dir,unsigned long address)

这些线入函数是用于获取与address相关联的pgd，pmd和pte项。页表查询从一个指向结构mm_struct的指针开始。与当前进程内存映射相关联的指针是current->mm。指向核心空间的指针由init_mm描述，它没有被引出到模块，因为它们不需要它。两级处理器定义pmd_offset(dir,add)为(pmd_t*)dir，这样就把pmd折合在pgd上。扫描页表的函数总是被声明为inline，而且编译器优化掉所有pmd查找。

unsigned long pte_page(pte_t pte)

这个函数从页表项中抽取物理页的地址。使用 pte_val(pte)并不可行，因为微处理器使用 pte 的低位存贮页的额外信息。这些位不是实际地址的一部分，而且需要使用 *pte_page* 从页表中、抽取实际地址。

pte_present(pte_t pte)

这个宏返回布尔值表明数据页当前是否在内存中。这是访问 pte 低位的几个函数中最常用的一个——这些低位被 *pte_page* 丢弃。有趣的是注意到不论物理页是否在内存中，页表始终在（在当前的 Linux 实现中）。这简化了核心代码，因为 *pgd_offse*t 及其它类似函数从不失败；另一方面，即使一个有零“ 驻留存贮大小 ”的进程也在实际 RAM 中保留它的页表。

仅仅看看这些列出的函数不足以使你对 Linux 的内存管理算法熟悉起来；实际的内存管理要复杂的多，而且还要处理其它一些繁杂的事，如高速缓存一致性。不过，上面列出的函数足以给你一个关于页面管理实现的初步印象；你可以从核心源码的 i*nclude/asm* 和 *mm* 子树中得到更好的信息。

虚拟内存区域

尽管换页位于内存管理的最低层，你在能有效地使用计算机资源之前还需要一些别的知识。核心需要一种更高级的机制处理进程看到它的内存方式。这种机制在 Linux 中以“ 虚拟内存区域的方式实现，我称之为“ 区域 ”或“ VMA ”。

一个区域是在一个进程的虚存中的一个同质区间，一个具有同样许可标志的地址的连续范围。它与“ 段 ”的概念松散对应，尽管最好还是将其描述为“ 具有自己属性的内存对象 ”。一个进程的内存映象由下面组成：一个程序代码（正文）区域；一个数据、BSS（未初始化的数据）和栈区域；以及每个活动的内存映射的区域。一个进程的内存区域可以通过查看 */proc/pid/maps* 看到。*/proc/self* 是*/proc/pid* 的特殊情况，它总是指向当前进程，做为一个例子，下面是三个不同的内存映象，我在#字号后面加了一些短的注释：

（代码 271）

每一行的域为：

start_end perm offset major:minor inode

---

* 事实中，在sparc上的这些函数并不是inline的，而是实际extern的函数，它们没有被引出到模块化的代码中。因此，你不能在运行在上的模块中使用这些函数，不过实际上一般也用不着那样做。

perm 代表一个位掩码包括读、写和执行许可；它表示对属于这个区域的页，允许进程做什么。这个域的最后一个字符要么是 p 表示私有的，要么是 s 表示共享的。

/proc//maps* 的每个域对应着结构 vm_area_struct 的一个域，我们将在下面描述这个结构。

实现 *mmap* 的方法的驱动程序需要填充在映射设备的进程地址空间中的一个 VMA 结构。因此，驱动程序的作者对 VMA 应该有个最起码的理解以便使用它们。

让我们看一下结构 vm_area_struct（在<linux/mm.h>）中最重要的几个域。这些域可能在设备驱动程序的 *mmap* 实现中被用到。注意核心维护 VMA 的列表和树以优化区域查找，vm_area_struct 的几个域被用来维护这个组织。VMA 不能按照驱动程序的意愿被产生，不然结构将会崩溃。VMA 的几个主要域如下：

unsigned long vm_start

unsigned long vm_end

一个 VMA 描述的虚地址介于 vma->vm_start 和 vma->vm_end 之间。这两个域是 */pro/*/maps* 中显示的最先两个域。

struct inode *vm_inode

如果这个区域与一个 inode 相关联（如一个磁盘文件或一个设备节点），这个域是指向这个 inode 的指针。不然，它为 NULL。

unsigned long vm_offset

inode 中这个区域的偏移量。当一个文件或设备被映射时，这是映射到这个区域的第一个字节的文件的位置（filp->f_ops）。

struct vm_operations_struct *vm_ops

vma->vm_ops 说明这个内存区域是一个核心“对象”，就象我们在本书中一直在用的结构 file。这个区域声明在其内容上操作的“方法”，这个域就是用来列出这些方法。

和结构 vm_area_struct 一样，vm_operations_struct 在<linux/mm.h>中定义；它包括了列在下面的操作。这些操作是处理进程内存需要的所有操作，它们以被声明的顺序列出。列出的原型是 2.0 的，与 1.2.13 的区别在每一项中都有描述。在本章的后面，这些函数中的部分会被实现，那时会更完全地加以描述。

void(*open)(struct vm_area_struct *vma);

在核心生成一个 VMA 后，它就把它打开。当一个区域被复制时，孩子从父亲那里继承它的操作，就区域用 vm->open 打开。例如，当 *fork* 将存在进程的区域复制到新的进程时，vm_ops->open 被调用以打开所有的映象。另一方面，只要 *mmap* 执行，区域在 file->f_ops->mmap 被调用前被产生，此时不调用 vm_ops->open。

void(*close)(struct vm_area_struct *vma);

当一个区域被销毁时，核心调用它的 *close* 操作。注意 VMA 没有相关的使用计数；区域只被打开和关闭一次。

void(*unmap)(struct vm_area_struct *vma,unsigned long addr,size_t len);

核心调用这个方法取消一个区域的部分或全部映射。如果整个区域的映射被取消，核心在 vm_ops->unmap 返回后立即调用 vm_ops->close。

void (*protect)(struct vm_area_struct *vma,unsigned long,size_t,unsigned int new prot);

当前未被使用。许可（保护）位的处理并不依赖于区域本身。

int(*sync)(struct vm_area_struct *vma,unsigned long,size_t,unsigned int flags);

这个方法被 *msync* 系统调用以将一个脏的内存区段保存到存贮介质上。如果成功则返回值为 0 ，如果有错，则返回一个负数。核心版本 1.2 让这个方法返回 void，因为这个函数不被认为会失败。

void(*advise)(struct vm_area_struct *vma,unsigned long,size_t,unsigned int advise);

当前未被使用。

unsigned long(*nopage)(struct vm_area_struct *vma,unsigned long address,int write_access);

当一进程试图访问属于另一个有效 VMA 的某页，而该页当前不在内存时，*nopage* 方法就会被调用，如果它为相关区域定义。这个方法返回该页的（物理）地址。如果这个方法不为这个区域所定义，核心会分配一个空页。通常，驱动程序并不实现 *nopage*，因为被一个驱动程序映射的区段往往被完全映射到系统物理地址。核心版本 1.2 的 *nopage* 具有一个不同的原型和不同的含义。第三个参数 write_access 被当做“不共享”——一个非零值意味着该页必须被当前进程所有，而零则表示共享是可能的。

unsigned long(*wppage)(struct vm_area_struct *vma,unsigned long address,unsigned long page);

这个方法处理“写保护”页面错，但目前不被使用。核心处理所有不调用区域特定的回调函数却往一个被保护的页面上写的企图。写保护被用来实现“写时拷贝（copy_on_write）”。一个私有的页可以被不同进程所共享，直到其中一个进程试图写它时。当这种情况发生时，页面被克隆，进程向自己的页拷贝上写。如果整个区域被称为只读，会有一 SIGSEGV 信息被发送给进程，写时拷贝就未能完成。

int (*swapout)(struct vm_area_struct *vma,unsigned long offset,pte_t *page_table);

这个方法被用来从交换空间取得一页。参数 offset 是相对区域而言（与上面 *swapout* 一样），而 entry 是页面的当前 pte——如果 *swapout* 在这一项中保存了一些信息，那么现在就可以用这些信息来取得该页。

一般说来，驱动程序并不需要去实现 *swapout* 或 *swapin*，因为驱动程序通常映射 I/O 内存，而不是常规内存。I/O 页是一些象访问内存一样访问的物理地址，但被映射到设备硬件而不是 RAM 上。I/O 内存区段或者被标记为“保留”，或者居于物理内存之上，因此它们从不被换出—交换 I/O 内存没什么实际意义。

内存映象

在 Linux 中还有与内存管理相关的第三个数据结构。VMA 和页表组织虚拟地址空间，而物理地址空间则由内存映象概括。

核心需要物理内存当前使用情况的一个描述。由于内存可以被看作是页面数组，因此这个信息也可以组织为一个数组。如果你需要其页面的信息，你就用其物理地址去访问内存映象。下面就是核心代码用来访问内存映象的一些符号：

typedef struct {/*…*/} mem_map_t

extern mem_map_t mem_map[];

映象本身是 mem_map_t 的一个数组。系统中的每个物理页，包括核心代码和核心数据，都在 mem_map 中有一项。

PAGE_OFFSET

这个宏表示由物理地址映射到的核心地址空间中的虚地址。PAGE_OFFSET 在任何用到“物理”地址的地方都必须要考虑。核心认为的物理地址实际上是一个虚拟地址，从实际物理地址偏移 PAGE_OFFSET——这个实际物理地址是在 CPU 外的电气地址线使用。在 Linux2.0.x 中，PAGE_OFFSET 在 PC 上都是零，在大多数其它平台上都不是零。2.1.0 版修改了 PC 上的实现，所以它现在也使用偏移映射。如果考虑到核心代码，将物理空间映射到高的虚拟地址有一些好处，但这已经超出了本书的范围。

int MAP_NR(addr)

当程序需要访问一个内存映象时，MAP_NR 返回在与 addr 关联的 mem_map 数组中的索引。参数 addr 可以是 unsigned long，也可以是一个指针。因为这个宏被几个关键的内存管理函数使用多次，所以它不进行 addr 的有效性检查；调用代码在必要的时候必须自己进行检查。

((nr<<PAGE_SHIFT)+PAGE_OFFSET)

没有标准化的函数或者宏可以将一个映象号转译为一个物理地址。如果你需要MAP_NR 的逆函数，这个语句可以使用。

内存映象是用来为每个内存页维护一些低级信息。在核心开发过程中，内存映象结构的准确定义变过几次，你不必了解细节，因为驱动程序不期望查看映象内部。

不过，如果你对了解页面管理的内部感兴趣的话，头文件<linux/mm.h>含有一大段注释解释mem_map_t 域的含义。

mmap 设备操作

内存映象是现代 Unix 系统中最有趣的特征之一。至于驱动程序，内存映射可以提供用户程序对设备内存的直接访问。

例如，一个简单 ISA 抓图器将图象数据保存在它自己的内存中，或者在 640KB-1KB 地址范围，或者在“ ISA 洞 ”（指 14MB-16MB 之间的范围参见第 8 章“ 硬件管理 ”中“ 访问设备板子上的内存 ”一节）中。

将图象数据复制到常规（并且更快）RAM 中是不定期抓图的合适的方法，但如果用户程序需要经常性地访问当前图象，使用 *mmap* 方法将更合适。

映射一个设备的意思是使用户空间的一段地址空间关联到设备内存上。当程序读写指定的地址范围时，它实际上是在访问设备。

正如你所怀疑的，并不是每个设备都适合 *mmap* 概念；例如，对于串口或其它面向流的设备来说它的确没有意义。*mmap* 的另一个限制是映射是以 PAGE_SIZE 为单位的。核心只能在页表一级处置虚地址，因此，被映射的区域必须是 PAGE_SIZE 的整数倍，而且居于页对齐的物理内存。核心通过使一个区段稍微大一点儿的办法解决了页面粒度问题。对齐的问题通过使用 vma->vm_offset 来处理，但这对于驱动程序并不可行——映射一个设备简化为访问物理页，它必须是页对齐的。

这些限制对驱动程序来说并不是很大的问题，因为不管怎样，访问设备的程序是设备相关的。它知道如何使得被映射的内存区段有意义，因此页对齐不是一个问题。当你 ISA 板子插到一个 Alpha 机器上时，有一个更大的限制，因为 ISA 内存是以 8 位、16 位或 32 位项的散布集合被访问的，没有从 ISA 地址到 Alpha 地址的直接映射。在这种情况下，你根本不能使用 *mmap*。不能进行 ISA 地址到 Alpha 地址的直接映射归因于两种系统数据传送规范的不兼容。Alpha 只能进行 32 位和 64 位的内存访问，而 ISA 只能进行 8 位和 16 位的传送，没有办法透明地从一个协议映射到另一个。结果是你根本不能对插在 Alpha 计算机的 ISA 板子使用 *mmap*。

当可行的时候，使用 *mmap* 有一些好处。例如，一个类似于 X 服务器的程序从显存中传送大量的数据；把图形显示映射到用户空间与 *lseek/write* 实现相比，显著地改善了吞吐率。另一个例子是程序控制 PCI 设备。大多数 PCI 外围设备都将它们的控制寄存映射到内存地址上，一个请求应用更喜欢能直接访问寄存器，而不是反复调用 *ioctl* 来完成任务。

mmap 方法是 file_oprations 结构的一部分，在 *mmap* 系统调用被发出时调用。在调用实际方法之前，核心用 mmap 完成了很多工作，因此，这个方法的原型与系统调用很不一样。这与其它调用如 *ioctl* 和 *select* 不同，它们在被调用之前核心并不做太多的工作。

系统调用如下声明（在 mmap（2）手册中有描述）：

mmap（caddr_t，size_t len，int prot，int flags，int fd，off_t offset）

另一方面，文件操作如下声明：

int （*mmap）（struct inode*inode,struct file*filp,struct vm_area_struct *vma）；

方法中 inode 和 filp 参数与第三章“ 字符设备驱动程序 ”中介绍的一样。vma 会有用以访问

设备的虚拟地址范围的信息。这样，驱动程序只需为这个地址范围构造合适的页表：如果需要，用一组新的操作代替 vma->vm_ops。

一个简单的实现

设备驱动程序的大多数 *mmap* 实现对居于周边设备上的某些 I/O 内存进行线性的映射。*/dev/mem* 和*/dev/audio* 都是这类重映射的例子。下面的代码来自 *drivers/char/mem.c*，显示了在一个被称为 simple（Simple Implementation Mapping Pages with Little Enthusiasm）的典型模块中这个任务是如何完成的：

（代码 277）

很清楚，操作的核心由 *remap_page_range* 完成，它被引出到模块化的驱动程序，因为它做了大多数映射需要做的工作。

维护使用计数

上面给出的实现的主要问题在于驱动程序没有维护一个与被映射区域的连接。这对*/dev/mem* 来说并不是个问题，它是核心的一个完整的部分，但对于模块来说必须有一个办法来保持它的使用计数是最新的。一个程序可以对文件描述符调用 *close*，并仍然访问内存映射的区段。然而，如果关闭文件描述符导致模块的使用计数降为零，那么模块可能被卸载，即使它们仍被通过 *mmap* 使用着。

试图关于这个问题警告模块的使用者是不充分的解决办法，因为可能使用 *kerneld* 装载和卸载你的模块。这个守护进程在模块的使用计数降为零时自动地去除它们，你当然不能警告 *kerneld* 去留神 *mmap*。

这个问题的解决办法是用跟踪使用计数的操作取代缺省的 vma->vm.ops。代码相当简单——用于模块化的*/dev/mem* 的一个完全的 *mmap* 实现如下所示：

（代码 278）

这个代码依赖于一个事实，即核心在调用 f_op->mmap 之前将新产生区域中的 vm_ops 域初始化为 NULL。为安全起见以防止在将来的核心发生什么改变，给出的代码检查了指针的当前值。

给出的实现利用了一个概念，即 *open（vma）*和 *close（vma）*都是缺省实现的一个补充。驱动程序的方法不须复制打开和关闭的内存区域的标准代码；驱动程序只是实现额外的管理。有趣的是注意到，VMA 的 *swapin* 和 *swapout* 方法以另外的方式工作——驱动程序定义的 vm_ops->swap*不是添加而是用完全不同的东西取代了缺省实现。

支持 mremap 系统调用

mremap 系统调用被应用程序用来改变映射区段的边界地址。如果驱动程序希望能支持 *mremap*，以前的实现就不能正确地工作，因为驱动程序没有办法知道映射的区域已经改变了。

Linux 的 *mremap* 实现不提醒驱动程序关于映射区域的改变。实际上，它到是通过 *unmap* 方法在区域减小时提醒驱动程序，但在区域变大时没有回调发出。

将减小告诉驱动程序隐含的基本思想法是驱动程序（或是将常规文件映射到内存的文件系统）需要知道区段什么时候被取消映射了，从而采取适应的动作，如将页面刷新到磁盘上。另一方面，映射区域的增大对驱动程序来说意义不大。除非调用 *mremap* 的程序访问新的虚地址。在实际情况中，映射从未使用的区段是很常见的（如未使用过的某些程序代码段）。因此，Linux 核心在映射区段增大时并不告诉驱动程序，因为 *nopage* 方法将会照管这些页。如果它们确实被访问了。

换句话说，当映射区段增大时，驱动程序未被提醒是因为 *nopage* 后来会这样做；从而不必在需要前使用内存。这个优化主要是针对常规文件的，它们使用真正的 RAM 进行映射。

因此，如果你想支持 *mremap* 系统调用，就必须实现 *nopage*。不过，一旦有了 *nopage*，你

可选择广泛地使用它，从而避免从 fops->mmap 调用 *remap_page_range*；这在下一个代码段中给出。在这个 *mmap* 的实现中，设备方法只取代了 vma->vm_fops。*nopage* 方法负责一次重映射一个页并返回其地址。

一个支持 *mremap*（为节省空间，不支持使用计数）的*/dev/mem* 实现如下所示：

（代码 279）

（代码 280）

如果 *nopage* 方法被留为 NULL，处理页面错的核心代码就将零页映射到出错虚地址。零页是一个写时拷贝页，被当作零来读，可以用来映射 BSS 段。因此，如果一个进程通过调用 *mremap* 扩展一个映射区段，并且驱动程序没有实现 *nopage*，你最终会得到一些零页，而不是段错。

注意，给出的实现远远不是最优的；如果内存方法能绕过 *remap_page_range* 而直接返回物理地址会更好。不幸的是，这个技术的正确实现牵涉到一些细节，只能在本章晚些时候搞清楚。而且上面给出的实现在核心 1.2 中并不能工作，因为 *nopage* 的原型在版本 1.2 和 2.0 之间做了修改。在本节中我不打算管 1.2 核心。

重映射特定的 I/O 区段

到目前为止，我们所看到的所有例子都是*/dev/mem* 的再次实现；它们将物理地址重映射到用户空间——或者至少这是它们认为它们所做的。然而，典型的驱动程序只想映射应用于它的外围设备的小地址区间，并非所有内存。

为了能为一个特定的驱动程序自定义*/dev/mem* 的实现，我们需要进一步来研究一下 *remap_page_range* 的内部。这个函数的完整原型是：

int remap_page_range(unsigned long virt_add，unsigned long phy_add,unsigned long size, pgprot_t prot);

这个函数的返回值通常为零或为一个负的错误代码。让我们看看它的参数的确切含义。

unsigned long virt_add

重映射开始处的虚拟地址。这个函数为虚地址空间 virt_add 和 virt_add+size 之间的范围构造页表。

unsigned long phys_add

虚拟地址应该映射到的物理地址。这个地址在上面提到的意义下是“ 物理的 ”这个函数影响 phys_add 到 phys_add+size 之间的物理地址。

unsigned long size

被重映射的区域的大小，以字节为单位。

pgprot_t prot

为新页所请求的“ 保护 ”。驱动程序不必修改保护，而且在 vma->vma_page_prot 中找到的参数可以不加改变地使用。如果你好奇，你可以在<Linux/mm.h>中找到更多的信息。

为了向用户空间映射整个内存区间的一个子集，驱动程序需要处理偏移量。下面几行为映射了从物理地址 simple_region_start 开始的 simple_region_size 字节大小的区段的驱动程序完成了这项工作：

（代码 281）

除了计算偏移量，上面的代码还为错误条件引入了两个检查。第一个检查拒绝将一个在物理空间未对齐的位置映射到用户空间。由于只有完整的页能被重映射，因此映射的区段只能偏移页面大小的整数倍。ENXIO 是这种情况下通常返回的错误代码，它被展开为“ 无此设备或地址 ”。

第二个检查在程序试图映射多于目标设备 I/O 区段可获得内存的空间时报告一个错误。代码中 psize 是在偏移被确定后剩下的物理 I/O 大小，vsize 是请求的虚存大小；这个函数拒绝映

射超出允许内存范围的地址。
注意，如果进程调用 *mremap*，它便可以扩展其映射。一个“非常炫耀”的驱动程序可能希望阻止这个发生；达到目的的唯一办法是实现一个 vma->nopage 方法。下面是这个方法的最简单的实现：

```
unsigned long simple_pedantic_nopage(struct vm_area_struct *vma,unsigned long address, int write_access);
{return 0;}        /*发送一个 SIGBUS*/
```

如果 *nopage* 方法返回 0 而不是一个有效的物理地址，一个 SIGBUS（总线错）被发送到当前进程（即发生页面错的进程）。如果驱动程序没有实现 *nopage*，进程在请求的虚地址处得到一个零页；这通常可以接受，因为 *mremap* 是个非常少用的系统调用，而且将零页映射到用户空间也没有安全问题。

重映射 RAM

在 Linux 中，物理地址的一页被标记在内存映象中是“保留的”，表明不被内存管理系统使用。例如在 PC 上，640KB 到 1MB 之间的部分被称为“保留的”，它被用来存放核心代码。*remap_page_range*的一个有趣的限制是，它只能给予对保留的页和物理内存之上的物理地址的访问。保留页被锁在内存中，是仅有的能安全映射到用户空间的页；这个限制是系统稳定性的基本要求*。
因此，*remap_page_range* 不允许你重映射常规地址——包括你通过调用 *get_free_page* 所获得的那些。不过，这个函数做了所有一个硬件驱动程序希望它做的，因为它可以重映射高 PCI 缓冲和 ISA 内存——包括第 1 兆内存和 15MB 处 ISA 洞，如果在第八章“1M 以上的 ISA 内存”中提到的改变发生了的话。另一方面，当对非保留的页使用 *remap_page_range* 时，缺省的 *nopage* 处理程序映射被访问的虚地址处的零页。
这个行为可以通过运行 *mapper* 看到。*mapper* 是在 O’Reilly 的 FTP 站点上提供的文件中 *misc_programs* 里的一个示例程序。它是个可以快速测试 *mmap* 系统调用的简单工具。*mapper* 根据命令行选项映射一个文件中的只读部分，并把映射的区段输出到标准输出上。例如，下面这个交互过程表明*/dev/mem* 不映射位于 64KB 地址处的物理页（本例中的宿主机是个 PC，但在别的平台上结果应该是一样的）：
（代码 283）
remap_page_range 对处理 RAM 的无能为力说明象 *scull*p 这样的设备不能简单地实现 *mmap*，因为它的设备内存是常规 RAM，而不是 I/O 内存。
有两个办法可以绕过 *remap_page_range* 对 RAM 的不可用性。一个是“糟糕”的办法，另一个是干净的。

使用预定位

糟糕的办法要为你想映射到用户空间的页在 mem_map[MAP_NR(page)]中置 PG_reserved 位。这样就预定了这些页，而一旦预定了，*remap_page_range* 就可以按期望工作了。设置标志的代码很短很容易，但我不想在这儿给出来，因为另一个方法更有趣。不用说，不释放页面之前，预定的位必须被清除。
有两个原因说明为什么这是个好办法。第一，被标为预定的页永远不会被内存管理所动。核心在数据结构初始化之前系统引导时确定它们，因此不能用在任何其它用途上。而另一方面，通过 *get_free_page,vmalloc* 或其它一些方式分配的页都是由内存子系统处理的。即使 2.0 核心在你运行时预定额外的页并不崩溃，这样做可能在将来会产生问题。因而是不鼓励的。不过你可以尝试这种快且脏的技术看看它是任何工作的。

---

* 当某页成为进程内存映象的一部分时，它的使用计数必须增加，因为在取消映射时，它会被减小。 这类锁定不能在活动的RAM页上实施，因为它可能阻止正常的系统操作（象swapping和allocation/deallocation）。

预定页不是个好办法的第二个原因是被预定的页不被算做是整个系统内存的一部分，有的用户在系统 RAM 发生变化时可能会很在意——用户经常留意空闲内存数量，而总的内存量一般总和空闲内存一道显示。

实现 *nopage* 方法

将实际 RAM 映射到用户空间的一个较好的办法是用 vm_ops->nopage 来一次处理一个页面错。作为 *scullp* 模块一部分的一个示例实现在第七章“把握内存”中介绍过。

scullp 是面向页的字符设备。因为它是面向页的，所以可以在它的内存中实现 *mmap*。实现内存映射的代码用了一些以前在“Linux 的内存管理”中介绍过的概念。

在查看代码之前，让我们看一下影响 *scullp* 中 *mmap* 实现的设计选择。

- 设备为模块更新使用计数
  在卸载模块时为了避免发生问题，内存区域的 *open* 和 *close* 方法被实现去跟踪模块的使用。
- 设备为页更新使用计数
  这是为保证系统稳定是一个严格要求；不能更新这个计数将导致系统崩溃。每个页有其自己的使用计数；当它降为零时，该页被插入到空闲页表。当一个活动映象被破坏掉时，核心会将相关 RAM 页的使用计数减小。因此，驱动程序必须增加它所映射的每个页的使用计数（注意，这个计数在 *nopage* 增加它时不能为零，因为该页已经被 fops->write 分配了）。
- 只要设备是被映射的，*scullp* 就不能释放设备内存
  这与其说是个要求，不如说是项政策，这与 *scull* 及类似设备的行为不同，因为它们在被因写而打开时长度被截为 0。拒绝释放被映射的 *scullp* 设备允许一个设备一个进程重写正被另一个进程映射的区段，这样你就可以测试并看到进程与设备内存之间是如何交互的。为避免释放一个被映射的设备，驱动程序必须保存一个活动映射的计数；设备结构中的 vma 域被用于这个目的。
- 只有在 *scullp* 的序号 order 参数为 0 时才进行内存影射
  这个参数控制 *get_free_pages* 是如何调用的（见第七章中“get_free_pages 和朋友们”一节）。这个选择是由 *get_free_pages* 的内部机制决定的——*scullp* 利用的分配机制。为了最大化分配性能，Linux 核心为每个分配的序号（order）维护一个空闲页的列表，在一个簇中只有第一页的页计数由 *get-free_pages* 增加和 *free_pages* 减少。如果分配序号大于 0，那么对一个 *scullp* 设备来说 *mmap* 方法是关闭的，因为 *nopage* 只处理单项，而不是一簇页。

最后一个选择主要是为了保证代码的简单。通过处理页的使用计数，也有可能为多页分配正确地实现 *mmap*，但那样只能增加例子的复杂性，而不能带来任何有趣的信息。

如果代码想按照上面提到的规则来映射 RAM，它需要实现 *open*，*close* 和 *nopage*，还要访问 mem_map。

scullp_mmap 的实现非常短，因为它依赖于 *nopage* 来完成所有有趣的工作：

（代码 285　#1）

开头的条件语句是为了避免映射未对齐的偏移和分配序号不为 0 的设备。最后，vm_ops->open 被调用以更新模块的使用计数和设备的活动映射计数。

open 和 *close* 就是为了跟踪这些计数，被定义如下：

（代码 285　#2）

由于模块生成了 4 个 *scullp* 设备并且也没有内存区域可用的 private_data 指针，所以 *open* 和 *close* 取得与 vma 相关联的 *scullp* 设备是通过从 inode 结构中抽取次设备号。次设备号被用

来从设备结构的 scullp_devices 数组取偏移后得到指向正确结构的指针。
大部分工作是由 *nopage* 完成的。当进程发生页面错时，这个函数必须取得被引用页的物理地址并返回给调用者。如果需要，这个方法可以计算 address 参数的页对齐。在 *scullp* 的实现中，address 被用来计算设备里的偏移；偏移又被用来在 *scullp* 的内存树上查找正确的页。
（代码 286　#1）
最后一行增加页计数；这个计数在 atomic_t 中生命，因此可以由一个原子操作更新。事实上，在这种特定的情况下，原子更新并不是严格要求的，因为该页已经在使用中，并且没有与中断处理程序或别的异步代码的竞争条件。
现在 *scullp* 可以按预期的那样工作了，正如你在工具 *mapper* 的示例输出中所看到的：
（代码 286　#2）
（代码 287　#1）

重映射虚地址

尽管很少需要重映射虚地址，但看看驱动程序如何用 *mmap* 将虚地址映射到用户空间是很有趣的。这里虚地址指的是由 *vmalloc* 返回的地址，也就是被映射到核心页表的虚地址。本节的代码取自 *scullv*，这个模块与 *scullp* 类似，只是它通过 *vmalloc* 分配存储。
scullv 的大部分实现与我们刚刚看到的 *scullp* 完全类似，除了不需要检查分配序号。原因是 *vmalloc* 一次只分配一页，因为单页分配比多页分配容易成功的多。因此，使用计数问题在通过 *vmalloc* 分配的空间中不适用。
scullv 的主要工作是构造一个页表，从而可以象连续地址空间一样访问分配的页。而另一方面，*nopage* 必须向调用者返回一个物理地址。因此，*scullv* 的 *nopage* 实现必须扫描页表以取得与页相关联的物理地址。
这个函数与我们在 *scullp* 中看到的一样，除了结尾。这个代码的节选只包括了 *nopage* 中与 *scullp* 不同的部分。
（代码 287　2#）

```
    atomic_inc(&mem_map[MAP_NR(page)]).count;
    return page;
  }
```

页表由本章开始时介绍的那些函数来查询。用于这个目的的页目录存在核心空间的内存结构 init_m 中。
宏 VMALLOC_VMADDR(pageptr)返回正确的 unsigned long 值用于 *vmalloc* 地址的页表查询。注意，由于一个内存管理的问题，这个值的强制类型转换在早于 2.1 的 X86 核心上不能工作。在 X86 的 2.1.1 版中内存管理做了改动，VMALLOC_VMADDR 被定义为一个实体函数，与在其它平台上一样。
最后要提到的一点是 init_mm 是如何被访问的，因为我前面提到过，它并未引出到模块中。实际上，*scullv* 要作一些额外的工作来取得 init_mm 的指针，解释如下。
实际上，常规模块并不需要 init_mm，因为它们并不期望与内存管理交互；它们只是调用分配和释放函数。为 *scullv* 的 *mmap* 实现很少见。本小节中介绍的代码实际上并不用来驱动硬件；我介绍它只是用实际代码来支持关于页表的讨论。
不过，既然谈到这儿，我还是想给你看看 *scullv* 是如何获得 init_mm 的地址的。这段代码依赖于这样的事实：0 号进程（所谓的空闲任务）处于内核中，它的页目录描述了核心地址空间。为了触到空闲任务的数据结构，*scullv* 扫描进程链表直到找到 0 号进程。
（代码 288）
这个函数由 fops->mmap 调用，因为 *nopage* 只在 *mmap* 调用后运行。

基于上面的讨论，你也许还想将由 *vremap*（如果你用 Linux2.1，就是 *ioremap*）返回的地址映射到用户空间。这很容易实现，因为你可以直接使用 *remap_page_range*，而不用实现虚拟内存区域的方法。换句话说，*remap_page_range* 已经可用以构造将 I/O 内存映射到用户空间的页表；并不需要象我们在 *scullv* 中那样查看由 *vremap* 构造的核心页表。

直接内存访问

直接内存访问，或 DMA，是我们内存访问方面讨论的高级主题。DMA 是一种硬件机制，它允许外围组件将 I/O 数据直接从（或向）主存中传送。

为了利用硬件的 DMA 能力，设备驱动程序需要能正确地设置 DMA 传送并能与硬件同步。不幸的是，由于 DMA 的硬件实质，它非常以来于系统。每种体系结构都有它自己管理 DMA 传送的技术，编程接口也互不相同。核心也不能提供一个一致的接口，因为驱动程序很难将底层硬件机制适当地抽象。本章中，我将描述 DMA 在 ISA 设备及 PCI 外围上是如何工作的，因为它们是目前最常用的外围接口体系结构。

不过，我不想讨论 ISA 太多的细节。DMA 的 ISA 实现过于复杂，在现代外围中并不常用。目前 ISA 总线主要用在哑外围接口上，而需要 DMA 能力的硬件生产商倾向于使用 PCI 总线。

DMA 数据传送的概况

在介绍编程细节以前，我们先大致看看 DMA 传送是如何工作的。为简化讨论，只介绍输入传送。

数据传送有两种方式触发：或者由软件请求数据（通过一个函数如 *read*），或者由硬件将数据异步地推向系统。

在第一中情况下，各步骤可如下概括：

- 当一个进程调用一个 *read*，这个驱动程序方法分配一个 DMA 缓冲区，并告诉硬件去传诵数据。进程进入睡眠。
- 硬件向 DMA 缓冲区写数据，完成时发出一个中断。
- 中断处理程序获得输入数据，应答中断，唤醒进程，它现在可以读取数据。

有时 DMA 被异步地使用。例如，一些数据采集设备持续地推入数据，即使没有人读它。这种情况下，驱动程序要维护一个缓冲区，使得接下来的一个 *read* 调用可以将所有累积的数据返回到拥护空间。这种传送的步骤稍有不同：

- 硬件发出一个中断，表明新的数据到达了。
- 中断处理程序分配一个缓冲区，告诉硬件将数据传往何处。
- 外围设备将数据写入缓冲区；当写完时，再次发出中断。
- 处理程序派发新数据，唤醒所有相关进程，处理一些杂务。

上面这两种情况下的处理步骤都强调：高效的 DMA 处理以来于中断报告。尽管可以用一个轮询驱动程序来实现 DMA，这样做并无意义，因为轮询驱动程序会将 DMA 相对于简单的处理器驱动 I/O 获得的性能优势都抵消了。

这里介绍的另一个相关问题是 DMA 缓冲区。为利用直接内存访问，设备驱动程序必须能分配一个特殊缓冲区以适合 DMA。注意大多数驱动程序在初始化时分配它们的缓冲区，一直使用到关机——因此，上面步骤中“分配”一词指的是“获得以前已分配的缓冲区”。

分配 DMA 缓冲区

DMA 缓冲区的主要问题是当它大于一页时，它必须占据物理内存的连续页，因为设备使用 ISA 或 PCI 总线传送数据，它都只携带物理地址。有趣的是注意到，这个限制对 Sbus 并不适用（见第 15 章“外围总线概览”中“Sbus”一节），它在外围总线上适用虚地址。

尽管 DMA 缓冲区可以在系统引导或运行时分配，模块只能在运行时分配其缓冲区。第七章

介绍了这些技术：“ Playing Dirty ”讲述在系统引导时分配；“ *kmalloc* 的真实故事 ”和“ *get_free_page* 和朋友们 ”讲述运行时分配。如果你用 *kmalloc* 或 *get_free_page*，你必须指定 GFP_DMA 优先级，与 GFP_KERNEL 或 GFP_ATOMIC 进行异或。

GFP_DMA 要求内存空间必须适合于 DMA 传送。核心保证能够进行 DMA 的缓冲区具有以下两个特点。第一，当 *get_free_page* 返回不止一页时，其物理地址必须是连续的（不过，一般情况下的确如此，与 GFP_DMA 无关，因为本身就是以成簇的连续页来组织空闲内存的）。第二，当 GFP_DMA 等设备时，核心保证只有低于 MAX_DMA_ADDRESS 的地址才被返回。宏 MAX_DMA_ADDRESS 在 PC 上被设为 16MB，用以对付马上就会讲到的 ISA 限制。

在 PCI 情况下，没有 MAX_DMA_ADDRESS 的限制，PCI 设备驱动程序在分配它的缓冲区时应避免设置 GFP_DMA。

自行分配

我们已经明白为何*get_free_page*(从而*kmalloc*)不能返回超过 128KB（或更一般地，32 页）的连续内存空间。但这个要求很容易失败，即使当分配的缓冲区小于 128KB时，因为随着时间的推移，系统内存会成为一些碎片*。

如果核心不能返回所需数量的内存，或如果你需要超过 128KB 的内存（例如对于一个 PCI 抓图器来说，这是个很常见的需求），相对于返回-ENOMEM 的一个办法是在引导时分配内存或为你的缓冲区保留物理 RAM 的顶端。我在第七章“ Playing Dirty ”中讲述了引导时的分配，但这个办法对模块不适用。保留 RAM 顶部可以通过向核心传递一个 mem=参数来完成。例如，如果你有 32M，参数 mem=31M 防止核心适用顶部一兆。你的模块以后可以用下面的代码来获得对该内存的访问：

```
dmabuf=vremap(0x1F00000 /*31MB*/, 0x100000 /* 1MB */ );
```

我自己分配DMA缓冲区的实现在*allocator.c*模块（和一个相配的头文件）中。你可以在*src/misc-modules*的示例文件中找到一个版本，最新版本总可以在我的FTP站点找到：*ftp://ftp.systemy.it/pub/develop*。你也可以找核心补丁*bigphysarea*，它和我的分配程序完成同样的工作。

总线地址

当进行 DMA 时，设备驱动程序必须与连在接口总线上的硬件对话，这里使用物理地址，但程序代码使用虚地址。

事实上，情况还要复杂一些。基于 DMA 的硬件使用总线地址，而不是物理地址。尽管在 PC 上，ISA 和 PCI 地址与物理地址一样，但并不是所有平台都是这样。有时接口总线是通过将 I/O 地址影射到不同物理地址的桥接电路被连接的。

Linux 核心通过引出定义在<*asm/io.h*>中的下列函数来提供一个可移植的解决方案。

```
unsigned long virt_to_bus(volatile void * address);
void * bus_to_virt(unsigned long address);
```

其中 virt_to_bus 转换在驱动程序需要向一个 I/O 设备（如一个扩展板或 DMA 控制器）发送地址信息时必须使用，而 bus_to_virt 在收到来自连于总线上的硬件地址信息时必须使用。

如果你查看依赖于前面讲的 allocator 机制的代码，你会发现这些函数的使用例子。这些代码也依赖于 *vremap*，因为上下文：

（代码 292）

尽管与 DMA 无关，我们值得再了解两个核心引出的函数：

```
unsigned long virt_to_phys(volatile void * address);
```

* “ 碎片 ”这个词一般用于磁盘，表示文件在磁介质上不是连续地存放。这个概念同样适用于内存，即当每个虚拟地址空间都散布在整个物理RAM，很难为DMA的缓冲区请求分配连续的空闲页。

void * phys_to_virt(unsigned long address);

这两个函数在虚地址和物理地址之间进行转换；它们在程序代码需要和内存管理单元（MMU）或其它连在处理器地址线上的硬件对话时被用到。在 PC 平台上，这两对函数完成同样的工作；但将它们分开是重要的，既为了代码的清晰，也为了可移植性。

ISA 设备 DMA

ISA 总线允许两类 DMA 传送：“ native DMA ”使用主板上的标准 DMA 控制器电路驱动 ISA 总线上的信号线；另一方面，“ ISA bus-master DMA ”则完全由外围设备控制。后一种 DMA 类型很少用，所以不值得在这里讨论，因为它类似于 PCI 设备的 DMA，至少从驱动程序的角度看是这样的。一个 ISA bus-master 的例子是 1542 SCSI 控制器，它的驱动程序是核心源码中的 *drivers/scsi/aba1542.c*。

关于 native DMA，有三个实体参与了 ISA 总线上的 DMA 数据传送：

8237 DMA 控制器（DMAC）

控制器存有 DMA 传送的信息，如方向、内存地址、传送大小。它还有一个跟踪传送状态的计数器。当控制器收到一个 DMA 请求信号，它获得总线控制并驱动信号线以使设备可以读写数据。

外围设备

设备在准备好传送数据时，必须激活 DMA 请求信号。实际的传送由 DMAC 控制；当控制器通知了设备，硬件设备就顺序地从/往总线上读/写数据。

设备驱动程序

驱动程序要做的很少，它向 DMA 控制器提供方向、RAM 地址、传送大小。它还与外围设备对话准备好传送数据或在 DMA 结束时响应中断。

原先在PC中使用的DMA控制器能管理 4 个通道，每个通道与一组DMA寄存器关联，因此 4 个设备可以同时在控制器中存储它们的DMA信息。新一些的PC有相当于两套DMAC的设备[*]：第二个控制器（主）连向系统处理器，第一个（从）连在第二个控制器的 0 通道上[+]。通道编号为 0~7；4 号通道对ISA外围不可用，因为它是内部用来将从控制器级联到主控制器上。这样从控制器上可用的通道为 0~3（8 位通道），主控制器上为 5~7（16 位通道[♣]）。每次DMA传送的大小存储在控制器中，是一个 16 位数，表示总线周期数。因此从控制器的最大传送大小为 64KB，主控制器为 128KB。

由于 DMA 是系统范围的资源，因此核心协助处理它。它用一个 DMA 注册项提供 DMA 通道的请求和释放机制，并用一组函数配置 DMA 控制器的通道信息。

注册 DMA 的使用

你应该已经熟悉核心注册项了——我们在 I/O 端口和中断线那里见过它们。DMA 通道的注册项与其它类似。在包含了<asm/dma.h>后，下面的函数可以用来获得和释放 DMA 通道的所有权：

int request_dma(unsigned int channel, const char *name);

void free_dma(unsigned int channel);

参数 channel 是 0 到 7 之间的一个数，或更精确地说，是一个小于 MAX_DMA_CHANNELS 的正数。在 PC 上，MAX_DMA_CHANNELS 被定义为 8，以匹配硬件。参数 name 是确定设备的一个字符串。指定的名字出现在文件/proc/dma 中，可由拥护程序读出。

---

[*] 这些电路现在是主板芯片组的一部分，但在几年前，它们是两个独立的 8237 芯片。

[+] 最初的PC只有一个控制器；第二个是在 286 平台上开始加上的。第二个控制器以主控制器的身份连接的原因是它能处理 16 位的传送，而第一个控制器一次只传送 8 位，它的存在只是为了后向兼容。

[♣] 一个总线I/O周期传送两个字节。

request_dma 在成功时返回 0，有错误时返回-EINVAL 或-EBUSY。前者表明请求的通道出了范围，后者表明有其它设备正占有这个设备。

我建议你对待 DMA 通道象对待 I/O 端口和中断线一样认真；在 *open* 时请求通道比从 init_module 中请求要好的多。推迟请求可以允许驱动程序间的一些共享；例如，你的声卡和你的模拟 I/O 接口可以共享 DMA 通道，只要它们不在同时使用。

我同时也建议你在请求中断线之后请求 DMA 通道，并在中断之前释放它。这是请求这两个资源的常规顺序；依照这个顺序可以避免可能的死锁。注意每个使用 DMA 的设备需要一个 IRQ 线，不然无法表明数据传送的完成。

在典型的情况下，*open* 的代码看起来如下，这是个虚设的 *dad* 模块（DMA 获取设备）。*dad* 设备使用一个快速的中断处理程序，不支持共享 IRQ 线。

（代码 295　#1）

与 *open* 匹配的 *close* 实现如下所示：

（代码 295　#2）

下面是小一个装有声卡的系统上*/proc/dma* 文件的内容：

```
merlino% cat /proc/dma
     1:  Sound Blaster8
     4:  cascade
```

有趣的是注意到缺省的声卡驱动程序在系统引导时获得 DMA 通道，并永不释放。显示的 cascade 项只是占据一个位置，表明通道 4 对驱动程序不可用，如前所述。

与 DMA 控制器对话

注册完成后，驱动程序的主要工作是为正确的操作来配置 DMA 控制器。这项工作并不简单，好在核心引出了所有典型驱动程序所需的函数。

在 *read* 或 *write* 被调用，或者在预备异步传送时，驱动程序都需要配置 DMA 控制器。第二种情况，任务在 *open* 时或在对一个 *ioctl* 命令响应时被执行，这依赖于驱动程序及其实现策略。这里给出的代码一般是由 *read* 或 *write* 设备方法调用。

本小节对 DMA 控制器内部给出一个快速的概览，这样你就可以理解这里介绍的代码。如果你想学更多，我鼓励你阅读<asm/dma.h>和一些介绍 PC 体系结构的硬件手册。特别地，我并不关注 8 位和 16 位数据传输的区别。如果你在为 ISA 设备板子写设备驱动程序，你应该在设备的硬件手册里查找相关信息。

必须装入控制器的信息由三项组成：RAM 地址，必须传送的原子项数目（以字节或字为单位），传送的方向。为了这个目的，下面的函数由<asm/dma.h>引出：

```
void set_dma_mode(unsigned int channel, char mode);
```

说明通道是从设备读（DMA_MODE_READ）还是向设备写（DMA_MODE_WRITE）。还有第三个模式，DMA_MODE_CASCADE，用来释放对总线的控制。级联是第一个控制器连到第二个控制器上的方法，但它也可以由真正的 ISA bus-master 设备使用。我在这里不想讨论 bus-master。

```
void set_dma_addr(unsigned int channel, unsigned int addr);
```

分配 DMA 缓冲区的地址。这个函数将 addr 的低 24 位存入到控制器。参数 addr 必须是个总线地址（见“总线地址”）。

```
void set_dma_count(unsigned int channel, unsigned int count);
```

分配要传送的字节数。参数 count 对 16 位通道仍以字节为单位；在这种情况下，这个数必须是个偶数。

除了这几个函数，还有一些必须用来处理 DMA 设备的杂务工具：

void disable_dma(unsigned int channel);

一个 DMA 通道可以在控制器内被关闭。在 DMAC 被配置之前，通道应该被关闭以防止不正确的操作（控制器通过 8 位数据传送编程，这样前面的函数都不能被原子地执行。）

void enable_dma(unsigned int channel);

这个函数告诉控制器这个 DMA 通道含有效数据。

int get_dma_residue(unsigned int channel);

驱动程序有时需要知道一个 DMA 传送是否结束了。这个函数返回尚待传送的字节数。如果成功传送完，则返回 0；如果控制器还在工作，返回值则不可预知（但不是 0）。这种不可预知性反映一个事实，即这个余数是个 16 位值，通过两个 8 位输入操作获得。

void clear_dma_ff(unsigned int channel);

这个函数清除 DMA flip-flop。flip-flop 用来控制对 16 位寄存器的访问。这些寄存器由两个连续的 8 位操作来访问，flip-flop 用来选中低字节（当它被清除时）或高字节（当它被置时）。flip-flop 在 8 位传输完后自动反转；在访问 DMA 积存器前必须清除一次 flip-flop。

用这些函数，驱动程序可以实现如下所示的一个函数来预备一个 DMA 传送：

（代码 297）

如下的函数用来检查 DMA 的成功完成：

```
int dad_dma_isdone(int channel)
{
    return(get_dma_residue(channel)==0);
}
```

剩下唯一要做的事就是配置设备板子。这个设备特定的任务通常包括读写几个 I/O 端口。设备在很多地方不同。例如，有的设备期望程序员告诉硬件 DMA 缓冲区有多大，有时驱动程序必须从设备中读出被硬写入的数值。为了配置板子，硬件手册是你唯一的朋友。

DMA 和 PCI 设备

DMA 的 PCI 实现比 ISA 上要简单的多。

PCI 支持多个 bus-master，而 DMA 就简化成 bus-mastering。需要读写主存的设备只需要简单地请求获得总线的控制，接着就可以直接控制电信号。PCI 的实现在硬件级更精巧，在设备驱动程序中更容易管理。

编写 PCI 上的 DMA 传送由下列步骤组成：

分配一个缓冲区

DMA 缓冲区在内存中必须是物理连续的，但没有 16MB 寻址能力的限制。一个 get_free_page 调用就足够了。不必在优先级中指定 GFP _DMA。如果你真的需要它，你可以转向（不鼓励）在前面“ 分配 DMA 缓冲区 ”中介绍过的更具进攻性的技术。

和设备对话

扩展设备必须被告知 DMA 缓冲区。这通常意味着将缓冲区的地址和大小写入几个设备积存器。有时，DMA 的大小由硬件设备决定，但这是设备相关的。传往 PCI 设备的地址必须是总线地址。

正如你所看到的，不存在为 PCI 设备编写的通用代码。一个典型的实现如下所示，但每个设备都不相同，配置信息量变化也很大。

（代码 298）

快速参考

本章介绍了与内存处理有关的下列符号。下面的列表不包括第一节中介绍的符号，因为其列表太大，而且那些符号在设备驱动程序中也很少用到。

#include <linux/mm.h>

所有与内存管理有关的函数和结构在这个头文件中定义。

int remap_page_range(unsigned long virt_add, unsigned long phys_add,
unsigned long size, pgprot_t prot);

这个函数居于 mmap 的核心，它将开始于物理地址 phys_add 的 size 字节映射到 virt_add。与虚拟空间相关联的保护位在 port 中指定。

#include <asm/io.h>

unsigned long virt_to_bus(volatile void * address);

void * bus_to_virt(unsigned int address);

unsigned long virt_to_phys(volatile void * address);

void * phys_to_virt(unsigned long address);

这些函数在虚拟和物理地址之间转换。必须使用总线地址来和外围设备对话，物理地址用来和 MMU 电路对话。

/proc/dma

这个文件包括 DMA 控制器中已分配通道的文本形式快照。基于 PCI 的 DMA 并不显示，因为各个板子独立工作，不必在 DMA 控制器中分配一个通道。

#include <asm/dma.h>

这个头文件定义了所有与 DMA 有关的函数和宏。要使用下面的符号就必须包含它。

int request_dma(unsigned int channel, const char * name);

void free_dma(unsigned int channel);

这些函数访问 DMA 注册项。在使用 ISA DMA 通道前必须注册。

void set_dma_mode(unsigned int channel, char mode);

void set_dma_addr(unsigned int channel, unsigned int addr);

void set_dma_count(unsigned int channel, unsigned int count);

这些函数用来将 DMA 信息置入 DMA 控制器。addr 是总线地址。

void disable_dma(unsigned int channel);

void enable_dma(unsigned int channel);

在配置时，DMA 通道必须关闭。这些函数改变 DMA 通道的状态。

int get_dma_residue(unsigned int channel);

如果驱动程序想知道 DMA 传送进行的如何，它可以调用这个函数，返回尚需传送的数据字节数。DMA 成功完成后，函数返回 0；如果还在传送中，这个值是不可预知的。

void clear_dma_ff(unsigned int channel);

DMA flip-flop 被控制器用来用 8 位操作来传送 16 位的值。在传送任何数值到控制器前必须将其清楚。

第十四章 网络驱动程序

我们已经讨论了字符设备和块设备驱动程序，接着要讨论的是迷人的网络世界。网络接口是Linux 设备中的第三标准类，这一章就是讲述它们是如何与核心的其余部分交互的。
网络接口并不象字符和块设备那样存在于文件系统。相反，它在核心层处理包的发送和接收，并不与进程中的某个打开的文件绑定在一起。
网络接口在文件系统中的角色就象被安装的块设备。一个块设备在 blk_dev 数组和其它核心结构中注册它的特征，接着按照要求通过它的 *request_fn* 函数“发送”和“接收”块。类似地，一个网络接口必须在特定的数据结构中注册自己，从而在与外部世界交换包时可以被调用。
安装的磁盘与包发送接口有几个重要的不同。首先，磁盘以一个结点的形式存在于*/dev* 目录，而网络接口并不在文件系统中出现。不过两者之间最大的不同在于：磁盘是被请求向核心发送一个缓冲区，而网络接口则是请求向核心推送进来的包。
Linux 的网络子系统被设计成完全协议无关的。这对网络协议（IP vs. IPX 或其它协议）和硬件协议（以太网 vs.令牌环等）都是如此。网络驱动程序和核心之间的交互一次处理一个网络包；这允许协议可以干净地对驱动程序隐`藏起来，而物理传输则可以对协议隐藏起来。
本章描述网络接口如何与核心的其它部分紧密合作，并给出一个基于内存的模块化的网络接口，称之为（你可能已经猜到了）*snull*。为简化讨论，这个接口使用以太网硬件协议并传送IP 包。通过 *snull* 获得的知识可以很好地应用于 IP 以外的协议，从以太网移到其它硬件协议只要求你对使用的物理协议有所了解。
snull 的另一个限制是它不能在 Linux1.2 中编译。再说一遍，这样做只是为了保持代码简单，并避免在 *snull* 中加入一些另人厌倦的条件。不过，本章将会提到与网络驱动程序相关的可移植性问题。
本章并不介绍 IP 的编号原则，网络协议，以及其它普通的网络概念。这个主题与驱动程序作者无关，而且以不到几百页的篇幅想对网络技术有一个令人满意的概述是不可能的。感兴趣的读者可以参考一些讲述网络问题的书。
在讨论网络设备之前，我想提醒你网络事务中的原子数据项被称做一个八元组（octet），由八个数据位组成。在本章中我都这样使用。网络文档从不使用术语“字节”。

Snull 如何设计

本节讨论与 *snull* 网络接口有关的一些设计概念。尽管这些信息可能显得用处并不大，但如果不理解它则可能在研究示例代码时遇到一些困难。
第一位的设计决定（也是最重要的）是示例接口不应绑定于任何实际硬件。实际接口不依赖于所传送的协议，*snull* 的这个限制并不影响本章给出的示例代码，因为它是协议无关的。IP 限制的唯一影响是地址分配----我们将位示例接口分配 IP 地址。

分配 IP 号码

snull 模块生成两个接口。这样的接口与简单的环回（loopback）并不一样，这里你从一个接口传送的包总是环回到另一个接口，而不是它自己。它看起来好象是你有两个外部链接，但实际上你的计算机只是应答自己。

不幸的是，这个效果并不能仅仅通过 IP 号码分配来达到，因为核心不会从接口 A 发送一个指向它自己接口 B 的包。相反，这时它会使用环回通道，从而根本不通过 *snull*。为了能建立一个通过 *snull* 接口的通信，源和目的地址必须在数据传送的时候修改一下。换句话说，从一个接口发出的包应能被另一个接口接收，但外出包的接收者不能被认为是本机。这也适用于收到包的源地址。
为了收到这种“隐藏环回”的效果，*snull* 接口反转一下源和目的地址的第三个八元组的最低位。其效果就是发向网络 A（连在接口 sn0 上）的包在 sn1 接口上好象是属于网络 B。
为了避免和太多的号码打交道，我们给用到的 IP 号码分配一些符号名：

- snullnet0 是一个连接在 sn0 接口上的一个 C 类网络。类似地，snullnet1 是连在 sn1 上的网络。这两个网络的地址仅在第三个八元组的最低位不同。
- local0 是分配给接口 sn0 的 IP 地址；它属于 snullnet0。与 sn1 相关联的地址是 local1。local0 和 local1 的第三和第四个八元组必须都不相同。
- remote0 是 snullnet0 中的一个主机，它的第四个八元组与 local1 相同。所有发向 remote0 的包在其 C 类地址被接口代码修改后将到达 local1。主机 remote1 属于 snullnet1，并且它的第四个八元组与 local0 相同。

snull 接口的操作见图 14-1，图中与接口相关联的主机名印在接口名旁边。
下面是几个可能的网络号码。一旦你把这几行写到*/etc/networks*，你就可以用名字来称呼这些网络。这些值是从保留私用的号码范围中选取的。

```
snullnet0     192.168.0.0
snullnet1     192.168.1.0
```

下面是写入*/etc/hosts* 的可能的主机号码：

```
192.168.0.88        local0
192.168.0.99        remote0
192.168.1.99        local1
192.168.1.88        remote1
```

(图 14-1 Page304)
不过如果你的计算机已经连到了一个网络上，那么一定要注意。你选择的号码有可能是实际的 Internet 或 intranet 的号码，把它们分配给你的接口可能会妨碍与真正主机的通信。而且，尽管我给出的这些号码不是实际的 Internet 号，但也有可能被你的私用网所适用，如果它处于防火墙之后的话。
不论你选择什么号码，你可以通过发出下面的命令来正确地设置接口：
（代码 304 #1）
到此为止，接口的“远”端已经可以到达了。下面的屏幕快照显示了我的主机是如何通过 *snull* 到达 remote0 和 remote1 的。
（代码 304 #2）
注意你不可能达到属于这两个网络的其它主机，因为在包的地址被改变并被接收到后，你的计算机会把它丢弃。

包的物理传送

至于数据传送，*snull* 属于以太网一类。示例代码使用了核心的以太网支持。这使我们不必去实现网络设备一些令人厌倦的细节。
我选择以太网是因为现存网络的主体----至少与工作站相连的这一部分-----都是基于以太网技术，不论是 10base2，10baseT，还是 100baseT。另外，核心还提供了对以太网设备的一般化

的支持，因此没有理由拒绝使用。以太网设备的优势如此明显，连 *plip* 接口（一类使用打印机端口的接口）都自称是以太网设备。
在 *snull* 中使用以太网设置的最后一个优势是你可以在接口上运行 *tcpdump*。不过，如果你想这样做，你需要把接口称做 eth*x*，而不是 sn*x*。*snull* 模块已经准备好将自己声明为 eth*x*。如果在 *insmode* 命令行中指定 eth=1，你就选择了这种行为。如果你忘了为 *snull* 请求 eth 命名，*tcpdump* 会拒绝倾倒这个接口，而是返回一个“ 未知的物理层类型 ”错。
snull 接口的另一个设计决定是只处理 IP 协议，本章的讨论也仅限于 IP。不过要注意，接口驱动程序本身并不依赖于它所处理的底层协议；网络驱动程序根本不查看它所传送的包。关于多协议传送将在后面的“ 非以太网包头 ”中详细介绍。
不过说实话，*snull* 还是查看包内容的，甚至还要修改它们，因为这是为保证代码工作要求的。代码修改每个 IP 包头的源，目的，以及校验和，但不检查它是否真地携带了 IP 信息。这种快而脏的数据修改会破坏非 IP 包。如果你想让 *snull* 处理其它协议，你必须修改这个模块的源码。不过，这种需求不太可能增长，因为每个拥有 Linux 盒的人都运行 IP，而其它协议则是可选的。

与核心相连

我们将通过拆解 *snull* 源码来看看网络驱动程序的结构。保证有几个驱动程序的源码在手边会很有助于你理解我们的讨论。我个人推荐 *loopback.c*，*plip.c*，以及 *3c509.c*，以逐渐增加的复杂性排序。有 *skeleton.c* 在手边也很有帮助，尽管这个示例驱动程序并不能真正运行。所有这些文件都居于核心源码树的 *drivers/net* 下。

模块加载

当一个模块被加载到运行的核心时，它要请求一些资源，并提供一些方便的功能；这已不再新鲜。另外请求资源的方式也不新鲜。驱动程序要探测它的设备及硬件位置（I/O 端口和 IRQ 线）----但并不注册它们----就象在第九章中断处理中“ 安装一个中断处理程序 ”一节中介绍的一样。网络驱动程序通过它的函数 *init_module* 进行注册的方法与字符或块设备驱动程序不一样。与请求一个主设备号不同，驱动程序为每个新检测到的接口在一个网络设备的全局列表中插入一个数据结构。
每个接口用一个 device 结构描述。sn0 和 sn1 这两个 *snull* 接口的结构如下所示：
（代码 306）
注意第一个域，既名字域指向一个静态缓冲区，它在加载时将被填充。通过这个方法，可以晚点儿选择接口名，下面会给出解释。如果你想在这个结构中使用一个显式缓冲区，如“ 01234567 ”，我要警告你那样可能导致代码不能可靠地工作。这是因为编译器会将两个重复的串折叠；因此你得到的会是一个缓冲区和两个指向它的缓冲区。而且，编译器有可能将常量串存在只读内存中，这显然不是你想要的。
在下节之前我不想完整描述结构 device，因为它是一个庞大的结构，太早地肢解它没有什么好处。我想在驱动程序中使用这个结构，并在每个域被使用时再解释它。
前面的代码显式地使用了 device 结构中的 name 和 init 域。name 是第一个域，含有接口名（识别接口的字符串）。驱动程序可以将接口名硬写在程序中，也允许动态赋值，其工作方式如下：如果名字的第一个字符是个空或者空格，那么设备注册项就使用第一个可用的 eth*n* 名。这样第一个以太网接口就被称做 eth0，其它的按序号类推。*snull* 接口则被缺省地称为 sn0 和 sn1。不过，如果在加载时指定 eth=1，那么 *init_module* 将使用动态赋值。缺省名由 *init_module* 给出：

（代码 307 #1）
init 域是个函数指针。任何时候当你注册一个设备时，核心要求驱动程序初始化自己。初始化就是指探测物理接口，用正确的数值填充 device 结构，下一节将给予描述。如果初始化失败，这个结构就不能被链入网络设备的全局列表。这种特别的设置的方法在系统引导时特别有效；每个驱动程序都试图注册它自己的设备，但只有确实存在的设备才被链入列表。这与字符和块设备驱动程序不同，它们被组织成一个两级树，由主设备号和次设备号索引。
由于真正的初始化在别的地方完成，*init_module* 要做的工作非常少，只需一句如下：
（代码 307 #2）

初始化每个设备

设备的探测在接口的 init 函数里完成，它通常被称做“ 探测 ”函数。init 收到的唯一的参数是一个指向正被初始化的设备的指针，其返回值是 0 或者一个负的错误代码----通常是-ENODEV。
对 *snull* 接口并没有进行实际的探测，因为它未绑定到任何硬件上。当你为一个实际的接口写实际的驱动程序时，探测字符设备的原则仍然适用：在使用 I/O 端口之前先检测它们，在检测期间不要向它写。另外，你还要避免在此时注册 I/O 端口和中断线。真正的注册应该推迟到设备打开时；这个非常重要，特别是当中断线被其它设备共享时。每次当别的设备触发中断线时，你的接口当然不希望被调用，而应简单地回答：不，它不是我的。
实际上，在加载时进行设备探测对 ISA 设备并不鼓励，因为这有可能很危险----ISA 体系结构在容错方面名声不佳。由于这个原因，大多数网络驱动程序在以模块的方式加载时拒绝为其硬件探测，核心也只探测第一个网络接口，在一个网络设备检测出来后不再进行任何硬件测试。通常 dev->base_addr----当前设备的 I/O 基地址----决定了要做什么：

- 如果 dev->base_addr 是一个有效的设备 I/O 地址，将不再探测其它 I/O 位置，而是使用这个值。如果这个值在加载时被赋值，这种情况就会发生。
- 如果 dev->base_addr 是 0，那么探测设备是可以接受的。拥护可以通过在加载时置这个 I/O 地址为 0 来请求探测。
- 其它情况下，不进行探测。核心使用 0xffe0 来阻止探测，但其实任何无效值都行。这需要依赖于驱动程序来无声地拒绝 base_addr 中的一个无效地址。一个模块应该缺省地置这个地址为无效值来防止不期望的探测。注意查看 PCI 设备总是安全的，因为它并不牵扯任何探测（见第 15 章，外围总线概述）

正如你可能已经注意到的，用一个加载时的设置来控制探测与我们在 *skull* 中使用的技术是一样的。
当从 dev->init 中退出时，dev 结构应该用正确的值填充。初始化例程的主要工作就是填充这个结构。幸运的是，核心通过函数 *ether_setup* 填充结构 device 负责了一些以太网的缺省设置。
snull_init 的核心是：
（代码 308）

```
        /* keep the default flags, just add NOARP */
        dev->flags           |=IFF_NOARP;
```

这段代码唯一不寻常的特征是在标志中设置 IFF_NOARP。这指明接口不能使用 ARP，即“ 地址解析协议 ”。ARP 是一个低级的以太网协议；每个真实的以太网接口都懂得 ARP，因此不需要设置这个标志。有趣的是注意到一个接口在没有 ARP 时仍能工作。例如，*plip* 接口就是没有 ARP 支持的以太网类接口，与 *snull* 相似。这个主题将在后面的“ 地址解析 ”中详细讨论，device 结构将在下一节肢解。

现在我想介绍结构 device 的另一个域 priv。它的作用类似与我们在字符设备驱动程序中用过的 private_data 指针。与 fops->private_data 不同的是，priv 指针是在初始化时分配，而不是在打开时，因为 priv 所指向的数据项包含有接口的统计信息。有一点很重要，就是统计信息要保证总是可用的，即使在接口宕掉时，因为用户可能在任何时候通过调用 *ifconfig* 来显式统计信息。在初始化时而不是打开时分配 priv 浪费的内存是无关紧要的，因为多数被探测到的接口保持一直在系统中运行。*snull* 模块为 priv 声明了一个数据结构 snull_priv。这个结构包含了结构 enet_statistics，它是存放接口信息的标准地方。
下面这几条 *snull_init* 中的语句分配 dev->priv：
（代码 309 #1）

模块卸载

当模块被卸载时没有什么特殊的事情发生。函数 *cleanup_module* 在释放了与私有结构相关的内存后，只需将接口从列表中取消即可。
（代码 309 #2）

模块化的和非模块化的驱动程序

尽管在对字符设备和块设备来说，模块化和非模块化的驱动程序并没有什么引人注意的区别，但对网络驱动程序来说，情况并非如此。
如果一个驱动程序做为主流 Linux 核心的一部分发行的话，它并不声明自己的 device 结构，而是使用在 *drivers/net/Space.c* 中声明的结构。*Space.c* 声明了所有网络设备的链表，即包括 plip1 一类驱动程序特定的结构，也包括通用目的的 eth 设备。以太网根本不关心它们的 device 结构，因为它们使用通用目的的结构。这种通用的 eth 设备结构声明 *ethif_probe* 为它们的 init 函数。程序员要想在主流核心中插入一个新的以太网接口只需要在 *ethif_probe* 中加入一个对驱动程序初始化函数的调用。另一方面，非 eth 驱动程序的作者需要在 *Space.c* 中插入它们的 device 结构。在两种情况下，如果驱动程序必须被链到核心，只需要修改源文件 *Space.c*。
在系统引导时，网络初始化代码循环遍历所有的 device 结构，调用它们的探测函数（dev->init），向它们传递一个指向设备本身的指针。如果探测函数成功了，*Space.c* 初始化 device 结构。这种设置驱动程序的方式允许渐增地将设备赋予名字 eth0，eth1，依次类推，而不需要改变每个设备的 name 域。
另一方面，当加载一个模块化的驱动程序时，它声明它自己的 device 结构（入我们在本章中已经看到的那样），即使它控制的接口是以太网接口。
好奇的读者可以查看 *Space.c* 和 *net_init.c* 来得到更多关于接口初始化的信息。这里对驱动程序设置的介绍只是为了强调 init 设备方法的重要性。如果一个驱动程序模块包含了预填好的设备结构，那么它将不适合主流核心的初始化技术，并且如果结构 device 中引入新的域，会使它变的不能向前兼容。

设备结构的细节

device 结构居于网络驱动程序的真正核心，值得完全的描述。第一次阅读本书的读者可以跳过本节，因为开始时不需要对这个结构有详细的理解。下面的列表描述所有的域，但主要目的是提供一个参考而不是要被记住。本章的其余部分在一个域被示例代码用到的时候会简单地描述一下，所以你不必不停地回头来参考本节。

结构 device 在结构上可以分为两个部分：“ 可见的 ”和“ 不可见的 ”。可见部分由那些在静态 device 结构中显式赋值的域组成，象前面给出的在 *snull* 中出现的两项。其余的域内部使用。有些被驱动程序访问（例如在初始化时被赋值的那些），而有些不能动。本章在版本 2.0.30 前都是完全的。

可见的头

结构 device 的第一部分由下列域组成，按序为：

char *name;

设备名。如果第一个字符是 0（NULL 字符）或空格，*register_netdev* 给它分配名字 eth*n*，n 取合适的值。

unsigned long rmem_end;

unsigned long rmem_start;

unsigned long mem_end;

unsigned long mem_start;

这些域存有设备使用的共享内存的开始和结束地址。如果设备有不同的发送和接收内存，那么 mem 域就用做发送内存，而 rmem 用做接收内存。mem_end 和 mem_start 可以在系统引导时在核心命令行指定，它们的值由 *ifconfig* 获取。Rmem 域在驱动程序以外不会被引用。一般地，end 域被设置成使得 end-start 为板上可用内存量。

unsigned long base_addr;

I/O 基地址。这个域，和前面的一样，在设备检测时被赋值。*ifconfig* 可以用来显示和修改当前值。base_addr 可以在系统引导或加载时在核心命令行显式赋值。

unsigned char irq;

被赋予的中断号。当接口被列出时 dev->irq 由 *ifconfig* 打印出来。这个值通常在引导或加载时被设置，以后可以用 *ifconfig* 修改。

unsigned char start;

unsigned char interrupt;

这些域是二进制标志。start 通常在设备打开时设置，在关闭时清楚。在接口准备号运行时它是非零。interrupt 是用来告诉代码的高层一个中断到达接口，并正在处理中。

unsigned long tbusy;

这个域表明“ 传送忙 ”。当驱动程序不能再接收新的包发送时（既所有的输出缓冲区都满了），它应该为非零。使用 long 类型而不是 char 是因为有时要使用原子的位操作以避免竞争条件。注意在核心 1.2，tbusy 的确是个八位的域，向后可移植的驱动程序应该注意这一点。原子的位操作在第九章的“ 使用锁变量 ”一节中介绍过。

struct device *next;

用来维护链表；任何驱动程序都不能动这个域。

int (*init)(struct device *dev);

初始化函数。这个域通常是 device 结构中显式列出的最后一个域。

隐藏的域

device 结构包含几个额外的域，通常在设备初始化时被赋值。这些域中的一些携带了接口的信息，一些存在只是为了方便驱动程序（也就是说，核心并不使用它们）；还有一些域，最引人注意的是一些设备方法，它们是核心和驱动程序的接口。

我想分别列为三组，与域的实际顺序无关，那并不重要。

接口信息

多数接口信息都由函数 *ether_setup* 来正确设置。以太网卡在大部分域都可以依赖这个通用目的的函数，但 flags 和 dev_addr 域是设备特定的，必须在初始化时显式地赋值。
一些非以太网的接口可以使用类似于 *ether_setup* 的助手函数。*driver/net/net_init.c* 引出 *tr_setup*(令牌环)和 *fddi_setup*。如果你的设备不属于这些类中的一种，你需要自己为所有的域赋值。

unsigned short hard_header_len;

“硬件包头长”。发送包头中 IP 头（或其它协议信息）之前那部分的八元组个数。对以太网接口来说，这个值是 14。

unsigned short mtu;

“最大传送单元”。在包传输时，这个域由网络层使用。以太网的 MTU 为 1500 个八元组。

__u32 tx_queue_len;

在设备传送队列中可以排队的最大祯数。*ether_setup* 将这个值设为 100，不过你可以改变它。例如，*plip* 使用 10 以避免浪费系统内存（*plip* 比实际的以太网接口吞吐率要低）。

unsigned short type;

接口的硬件类型。这个域被 ARP 使用以判断接口支持的硬件地址类型。以太网接口把它设为 ARPHRD_ETHER----*ether_setup* 为你做这件事。

unsigned char addr_len;

unsigned char broadcast[MAX_ADDR_LEN];

unsigned char dev_addr[MAX_ADDR_LEN];

以太网地址长为六个八元组（我们是指接口板的硬件标志），播送地址由六个 0xff 八元组组成；*ether_setup* 负责这些值的正确设置。另一方面，设备地址必须以设备特定的方式从接口板中读出，驱动程序应把它复制到 dev_addr。这个硬件地址用来在把包交给驱动程序传送前产生正确的以太网包头。*snull* 并不使用物理接口，它生成一个它自己的物理地址。

unsigned short family;

接口的地址族，通常为 AF_INET。接口并不常查看这个域或者向其赋值。

unsigned short pa_alen;

协议地址长。对 AF_INET 来说为四个八元组。接口不需要修改这个数。

unsigned long pa_addr;

unsigned long pa_brdaddr;

unsigned long pa_mask;

刻划接口的三个地址：接口地址，播送地址，及网络掩码。这些值是协议特定的（既它们是“协议地址”）；如果 dev->family 是 INET，则它们为 IP 地址。这些域由 *ifconfig* 赋值，对驱动程序是只读的。

unsigned long pa_dstaddr;

plip 和 ppp 一类点到点协议使用这个域记录连接另一侧的 IP 号码。和前面的域一样，它也是只读的。

unsigned short flags;

接口标志。这个域含有下列位值。前缀 IFF 意为接口标志（InterFace Flags）。有些标志由核心管理，有些则是在初始化时由接口设置，以确认接口的能力。有效的标志是：

IFF_UP

当接口是活跃的时，核心置上该标志。这个标志对驱动程序是只读的。

IFF_BROADCAST

这个标志表明接口的播送地址是有效的。以太网卡支持播送。

IFF_DEBUG

查错模式。这标志控制 *printk* 调用的唠叨，还用在其它一些查错目的。尽管目前没有官方驱动程序使用它，用户程序可以通过 *ioctl* 来对其置位或者清除，你的驱动程序可以使用它。*misc-progs/netifdebug* 程序可以用来将这个标志打开或关闭。

IFF_LOOPBACK

这个标志在环回接口中要被置位。核心检测这个标志而不是将名字 lo 作为特殊接口硬写入程序。

IFF_POINTOPOINT

点到点的初始化函数应置位这个标志。例如，*plip* 对它置位。*ifconfig* 工具也可以对其置位和清除。当它被置位时，dev->pa_dstaddr 应该指向连接的另一端。

IFF_NOARP

常规网络接口可以传送 ARP 包。如果接口不能进行 ARP，它必须置这个标志。例如，点到点接口并不需要运行 ARP，它只能增加额外的通信，却不能获取任何有用的信息。*snull* 不具有 ARP 能力，因此它要对其置位。

IFF_PROMISC

这个标志被置位以获得杂类操作。在缺省情况下，以太网接口使用硬件过滤器以保证它只收到播送包和指向其硬件地址的包。而象 *tcpdump* 一类包监视器则在接口上设置杂类模式，以获取经过接口传输介质的所有包。

IFF_MULTICAST

能进行选播传送的接口要置这个标志。*ether_setup* 在缺省情况下对其置位。所以如果你的驱动程序不支持选播，它必须在初始化时清除这个标志。

IFF_ALLMULTI

这个标志告诉接口接收所有的选播包。只有当 IFF_MULTICAST 被置位，而主机由进行选播路由时，核心对其置位。它对接口时只读的。IFF_MULTICAST 和 IFF_ALLMULTI 早在 1.2 版就已经定义了，但那时并未使用。在后面“选播”一节我们将看到它是如何使用的。

IFF_MASTER

IFF_SLAVE

这些标志被加载均衡代码使用。接口驱动程序不需要知道它们。

IFF_NOTRAILERS

IFF_RUNNING

这些标志在 Linux 中不使用，只是为了和 BSD 兼容而存在。

当一个程序改变 IFF_UP，*open* 和 *close* 方法会被调用。当 IFF_UP 或其它标志被修改时，*set_multicast_list* 方法被调用。如果驱动程序因为标志的修改而要执行一些动作，那么必须在 *set_multicast_list* 中进行。例如，当 IFF_PROMIS 被置位或清除时，板上硬件过滤器必须被通知。这个设备方法的责任将在后面的“选播”一节简单介绍。

设备方法

与字符设备和块设备的情况一样，每个网络设备要声明在其上操作的函数。可以在网络接口上进行的操作列在下面。一些操作可以留为 NULL，还有一些通常不去动它们，因为 *ether_setup* 给它们分配合适的方法。

一个网络接口的设备方法可以分为两类：基本的和可选的。基本的包括那些为访问接口所需要的；可选的方法实现一些并不严格要求的高级功能。下面是基本方法：

```
int (*open)(struct device *dev);
```

打开接口。只要 *ifconfig* 激活一个接口，它就被打开了。*open* 方法要注册它需要的所有资源（I/O 端口，IRQ，DMA，等），打开硬件，增加模块的使用计数。

int (*stop)(struct device *dev);

终止接口。接口在关闭时就终止了；在打开时进行的操作应被保留。

int (*hard_start_xmit)(struct sk_buff *skb, struct device *dev)；

硬件开始传送。这个方法请求一个包的传送。这个包含在一个套接字缓冲区结构（sk_buff）中。套接字缓冲区在下面介绍。

int (*rebuild_header)(void *buf, struct device *dev, unsigned long raddr, struct sk_buffer *skb);

这个函数用来在一个包传送之前重构硬件包头。这个以太网设备使用的缺省包头用 ARP 向包中填入缺少的信息。*snull* 驱动程序实现了它自己的这个方法，因为 ARP 并不在 sn 接口上运行。（在本章的后面会介绍 ARP。）这个方法的参数是一些指针，分别指向硬件包头，设备，“ 路由器地址 ”（包的初始目的地），以及被传送的缓冲区。

int (*hard_header)(struct sk_buffer *skb, struct device *dev, unsigned short type,
void *daddr, void *saddr, unsigned len);

硬件包头。这个函数用以前获取的源和目的地址构造包头；它的任务是组织那些以参数的形式传给它的信息。*eth_header* 是以太网类接口的缺省函数， *ether_setup* 相应地对这个域赋值。给出的参数顺序适用于核心 2.0 或更高版本，但与 1.2 有所不同。这个改变对以太网驱动程序是透明的，因为它继承了 *eth_header* 的实现；其它驱动程序可能要处理一下这个不同，如果它们想保持向后兼容的话。

struct enet_statistics * (*get_stats)(struct device *dev);

当应用希望获得接口的统计信息时需要调用这个方法，例如，当运行 *ifconfig* 或 *netstat –i* 时。在 *snull* 中的一个示例实现将在后面 “ 统计信息 ” 中介绍。

int (*set_config)(struct device *dev, struct ifmap *map);

改变接口的配置。这个方法是配置驱动程序的入口点。设备的 I/O 地址和中断号可以在运行时用 *set_config* 改变。在接口不能探测到时，系统管理员可以适用这个能力。这个方法在后面的 “ 运行时配置 ” 中介绍。

其余的设备方法是被我称为可选的那些。传递给其中一些的参数在 Linux1.2 到 Linux2.0 的转变中改了好几次。如果你想写一个可以在两个版本核心都工作的驱动程序，你可以只为从 2.0 开始的版本实现这些操作。

int (*do_ioctl)(struct devices *dev, struct ifreg *ifr, int cmd);

执行接口特定的 *ioctl* 命令。这些命令的实现在后面的 “ 自定义 *ioctl* 命令 ” 中描述。这里给出的原形在 1.2 以上的核心都能工作。如果接口不需要任何接口特定的命令，那么结构 device 中相应的域可以留为 NULL。

void (*set_multicast_list)(struct device *dev);

当设备的选播列表改变和标志改变时，将调用这个方法。这里的参数传递与 1.2 版本不同。更多的细节和一个示例实现见 “ 选播 ” 一节。

int (*set_mac_address)(struct device *dev, void *addr);

如果接口支持改变硬件地址的能力，可实现这个函数。多数接口要么不支持这个能力，要么使用缺省的 *eth_mac_addr* 实现。这个原形与 1.2 版也不同。

#define HAVE_HEADER_CACHE

void (*header_cache_bind)(struct hh_cache **hhp, struct device *dev, unsigned short htype, __u32 daddr);

void (*header_cache_update)(struct hh_cache *hh, struct device *dev, unsigned char *haddr);

这些函数和宏在 Linux1.2 中没有。以太网驱动程序不必关心 header_cache 的问题，因为 *eth_setup* 会安排使用缺省的方法。

#define HAVE_CACHE_MTU

int (*change_mtu)(struct device *dev, int new_mtu);

如果接口的 MTU（最大传送单元）发生了改变，这个函数负责采取动作。这个函数和宏在 Linux1.2 中都没有。当 MTU 改变时，如果驱动程序要做一些特殊的事情，它应该声明它自己的函数，不然将由缺省函数来完成。如果你感兴趣，*snull* 有一个这个函数的模版。

工具域

其余的结构 device 中的域被接口用来保存一些有用的状态信息。其中一些被 *ifconfig* 和 *netstat* 用来向用户提供当前配置的信息。因此，接口应该对这些域赋值。

unsigned long trans_start;

unsigned long last_rx;

这两个域用来保存一些瞬间值。它们目前不用，但核心有可能将来使用这些计时提示。驱动程序负责在传送开始时和收到包时更新这些值。trans_start 域还可以被驱动程序用来检测锁定。驱动程序可以在等待一个“传送完成”的中断时用 trans_start 来检查超时。

void *priv

等价于 filp->private_data。驱动程序拥有这个指针，可以随意使用。通常这个私有数据结构含有一个 enet_statistics 结构项。这个域在以前的“初始化每个设备”中用过。

unsigned char if_prot;

这个域用来记录哪个硬件端口被接口使用（例如，BNC，AUI，TP）。任何数值都可以按需要赋给它。

unsigned char dma;

被接口使用的 DMA 通道。这个域被 *ioctl* 的 SIOCGIFMAP 命令使用。

struct dev_mc_list *mc_list;

int mc_count

这两个域被用来处理选播传送。Mc_count 是 mc_list 中项的个数。更多的细节见“选播”。

结构 device 中还有一些别的域，但驱动程序没有使用它们。

打开和关闭

我们的驱动程序可以在模块加载和核心引导时探测接口。下一步是给接口赋一个地址，这样驱动程序就可以通过它交换数据了。打开和关闭一个接口由 *ifconfig* 命令完成。

当使用 *ifconfig* 为一个接口赋地址时，它完成两项工作。第一，它通过 ioctl(SIOCSIFADDR)(即 Socket I/O Control Set InterFace ADDRess)来赋地址。接着它通过 ioctl(SIOCSIFFLAGS)(即 Socket I/O Control Set InterFace FLAGS) 对 dev->flag 中的 IFF_UP 置位来打开接口。

至于设备，ioctl(SIOCSIFADDR) 设置 dev->pa_addr，dev->family，dev->pa_mask，dev->pa_brdaddr，没有驱动程序函数被调用----这个任务是设备无关的，由核心来完成。不过，后一个命令 ioctl(SIOCSIFFLAGS)为设备调用 *open* 方法。

类似地，当一个接口关闭时，*ifconfig* 使用 ioctl(SIOCSIFFLAGS)来清除 IFF_UP，并且调用 *stop* 方法。
两个设备方法在成功时都返回 0，发生错误时，通常返回一个负值。
至于代码，驱动程序必须执行与字符和块设备同样的工作。*open* 请求它所需要的所有的系统资源，并告诉接口启动；*stop* 则关闭接口，并释放系统资源。
如果驱动程序不准备使用共享中断（例如，它不打算与旧的核心兼容），还有最后一步需要做。核心引出一个 irq2dev_map 阵列，它由 IRQ 号寻址，持有空指针；驱动程序也许想用这个数组将中断号映射到指向 device 结构的指针。这是在不使用接口处理程序的情况下，在一个驱动程序里支持一个以上接口的唯一方法。
另外，在接口可以和外界通信以前，硬件地址还必须从板上复制到 dev->dev_addr。硬件地址可以按驱动程序的意愿在探测时或打开时被赋值。*snull* 软件接口时从 *open* 里对其赋值；它用两个 ASCII 串伪造一个硬件号码。地址的第一个字节是个空字符（在后面的“ 地址解析 ”中解释）。
结果得到的 *open* 代码如下所示：
（代码 319）
（代码 320 #1）
正如你所看到的，device 结构中的几个域被修改了。start 表明接口已准备好， tbusy 断言发送者不忙（也就是说，核心可以发出一个包）。
stop 方法是 *open* 的操作的反转。由于这个原因，实现 *stop* 的函数通常调用 *close*。
（代码 320 #2）

包发送

网络接口执行的最重要的工作是数据发送和接收。我准备从发送开始，因为它相对比较简单。
当核心需要发送一个数据包时，它调用 *hard_start_transmit* 方法将数据放到一个输出队列。核心处理的每个包包含在一个套接字缓冲区结构（struct sk_buff）中，其定义见 <linux/skbuff.h>。这个结构从 Unix 用来表示一个网络连接的抽象，即套接字得名。即使接口与套接字无关，每个网络包在较高的网络层中一定属于某个套接字，任何套接字的输入输出缓冲区都是 sk_buff 结构的列表。同样的 sk_buff 结构在整个 Linux 网络子系统中都被用来承载网络数据，但在考虑接口时，一个套接字缓冲区就是一个包。
指向 sk_buff 的指针通常被称做 skb，我将在示例和正文中都使用使用这个习惯。
套接字缓冲区是一个复杂的结构，核心提供一组函数来对其操作。这些函数在后面的“ 套接字缓冲区 ”中描述----目前，知道 sk_buff 的一些基本事实已足以写出可工作的驱动程序。另外，我习惯于在扎入另人讨厌的细节之前先弄明白是如何工作的。
传递给 *hard_start_xmit* 的套接字缓冲区含有物理包，它具有传输层的包头。接口不需要修改被发送的数据。skb->data 指向被发送的包，skb->len 是它的长度，以八元组为单位。
snull 的包发送代码如下所示；物理发送机制被隔离在另一个函数中，因为每个接口驱动程序必须按照被驱动的特定硬件来实现它。
（代码 321）
这样发送函数只进行一些清晰的对包的检查，并通过硬件相关的函数发送数据。在一个中断表明一个“ 发送结束 ”的条件时，dev->tbusy 被清除。

包接收

从网络中接收数据比发送要复杂一些，因为必须分配一个 sk_buff，并从一个中断处理程序中将其交递给高层----接收包的最好的办法是通过中断，除非接口是象 *snull* 一样是纯软件的，或是环回接口。尽管有可能写轮询的驱动程序，而且在正式的核心里也的确有几个，但中断驱动的要好的多，不管是在数据吞吐率还是计算需求上。由于绝大多数网络接口都是中断驱动的，我不打算谈论轮询实现，它只是利用了核心计时器。

snull 的实现是将硬件细节和设备无关的工作分离开的。这样，在硬件收到一个包后，*snull_rx* 被调用，它已经在计算机的内存中了。*snull_rx* 因此收到一个指向数据的指针和包的长度。。这个函数唯一的责任就是将包和一些额外信息发送到网络代码的高层。其代码与数据指针及长度获得的方法无关。

（代码 322）

这个函数足够通用，可以作为任何网络驱动程序模版，但在你有信心重用这个代码段之前还需要一些解释。

注意缓冲区分配函数需要知道数据长度。者避免了在调用 *kmalloc* 时浪费内存。*dev_alloc_skb* 以原子优先级调用分配函数，因此它也可以在中断时安全地使用。核心还提供了套接字缓冲区分配的其它一些接口，但不值得在这里介绍；套接字缓冲区在本章后面的“ 套接字缓冲区 ”中详细介绍。

一旦有了一个有效的 skb 指针，就可以通过调用 *memcpy* 将包数据复制到这个缓冲区。*skb_put* 更新缓冲区中数据尾的指针，并返回一个指向新生成空间的指针。

不幸的是，包头中没有足够的信息来正确处理网络层----在缓冲区向上层传递之前，dev 和 protocol 域必须被赋值。接着我们需要指定如何执行校验和（*snull* 不进行任何校验和）。skb->ip_summed 可能的策略为：

CHECKSUM_HW

板子用硬件执行校验和。一个硬件校验和的离子是 Sparc HME 接口。

CHECKSUM_NONE

校验和完全有软件完成。对新分配的缓冲区，这是缺省的策略。

CHECKSUM_UNNECESSARY

不做任何校验和。这是 *snull* 和环回接口的策略。

在 1.2 核心版本中没有校验和选项和 ip_summed。

最后，驱动程序更新它的统计计数器记录一个新包被收到了。统计结构有几个域组成，最重要的是 rx_packets 和 tx_packets，它们包含收到的和发送的包的个数。所有的域在后面的“ 统计信息 ”中给出一个彻底的描述。

包接收的最后一步由 *netif_rx* 完成，它将套接字缓冲区递交到上一层。

中断驱动的操作

大多数硬件接口以中断处理程序的方式控制。接口中断处理器表明两种事件中的一种：一个新包到达了或一个包发送完成了。这种一般化并不是总适用，但它基本上揭示了与异步包传送相关的问题。PLIP 和 PPP 是不适用这种一般化的例子。它们处理同样的事件，但低级中断处理略有不同。

一般的中断例程可以通过检查在硬件设备上的一个状态寄存器来分辨新包到达中断与完成发送的通知。*snull* 接口工作方式类似，但其状态字在 dev->priv 中。网络接口的中断处理程序看起来如下：

（代码 324）

处理程序的第一个任务是接收一个指向正确的 device 结构的指针。你可以用 irq2dev_map[]（假如你在打开是给它赋了一个值）或者接收到的 dev_id 指针作为一个参数。如果你希望驱动程序可以与新于 1.3.70 的核心工作，你必须使用 irq2dev_map[]，因为早期版本中没有 dev_id。
这个处理程序中有趣的部分是处理“ 发送完成 ”的部分。接口通过清除 dev->tbusy 并标志网络下半部例程来相应发送完成。如果 *net_bh* 的确运行了，它会试图发送所有等待的包。
另一方面，包接收并不需要任何特殊的中断处理。所有需要做的就是调用 *snull_rx*。
实际上，当 *netif_rx* 被接收函数调用时，它所进行的实际操作只有标志 *net_bh*。换句话说，核心在一个下半部处理程序中完成了所有网络相关的工作。因此，网络驱动程序应该总是宣称它的中断处理程序太慢，因为下半部将会更早地执行（见第九章中“ 下半部设计 ”）。

套接字缓冲区

我们已经讨论了于网络接口相关的多数内容。下面几节我们将更细地讨论 sk_buff 时如何设计的。这几节既介绍这个结构的主要域，也介绍在套接字缓冲区上操作的函数。
尽管并没有理解 sk_buff 内部的严格需要，但是如果能理解它的内容将会有助于你解决问题和优化代码。例如，如果你看了 *loopback.c*，你会发现一个基于 sk_buff 内部知识的优化。
我不打算在这里描述整个结构，而只是那些可能被驱动程序用到的域。如果你想知道更多，你可以看<linux/skbuff.h>，结构的定义和函数的原形都在那里定义。至于这些域和函数如何使用的细节可以通过浏览核心源码得到。

重要的域

出于我们的目的，结构里重要的域是那些驱动程序的作者可能要用到的域。它们如下所示，无特别顺序。

struct device *dev;
: 设备接收或者发送这个缓冲区。

__u32 saddr;
__u32 daddr;
__u32 raddr;
: 源地址，目的地址，和路由器地址，由 IP 协议使用。raddr 是包要到达其目的地的第一步。这些域在包被发送前被设置，收到之后就不必赋值了。到达 *hard_start_xmit* 方法的外出包已经有了一个合适的硬件包头设置反映了“ 第一步 ”信息。

unsigned char *head;
unsigned char *data;
unsigned char *tail;
unsigned char *end;
: 这些指针用来访问包中的数据。head 指向分配空间的开始，data 是有效八元组的开始（通常比 head 略大），tail 是有效八元组的结束， end 指向 tail 可以到达的最大地址。观察它们的另一个方法是：可用缓冲区空间为 skb->end-skb->head，当前使用的数据空间为 skb->tail-skb->data。这种处理内存区域的清晰方法在 1.3 开发时才实现。这是 *snull* 没有被移植在 Linux1.2 上编译的主要原因。

unsigned long len;
: 数据本身的长度（skb->tail-skb->head）。

unsigned char ip_summed;

这个域有驱动程序对进来包设置，由 TCP/UDP 校验和使用。它在前面的“包接收”中介绍过。

unsigned char pkt_type;

这个域被内部用来发送进来包。驱动程序负责将其设置为 PACKET_HOST（这个包是我的），PACKET_BROADCAST，PACKET_MULTICAST，或是 PACKET_OTHERHOST（不，这个包不是我的）。以太网驱动程序并不显式地修改 pkt_type，因为 eth_type_trans 会为它做这件事。

union { unsigned char *raw; […]} mac;

与 pkt_type 类似，这个域被用来处理进来包，必须在包接收时设置。函数 eth_type_trans 为以太网驱动程序负责这件事。非以太网驱动程序应设置 skb->mac.raw 指针，后面“非以太网包头”中将会提到。

结构中其余的域并无特别兴趣。它们被用来维护缓冲区列表，解释占有缓冲区的套接字的内存，等等。

在套接字缓冲区上操作的函数

使用 sock_buff 的网络设备通过正式的接口函数在这个结构上操作。有很多在套接字缓冲区上操作的函数，下面是最有趣的一些：

struct sk_buff *alloc_skb(unsigned int len, int priority);

struct sk_buff *dev_alloc_skb(unsigned int len);

分配一个缓冲区。*alloc_skb* 分配一个缓冲区并初始化 skb->data 和 skb->tail 到 skb->head。*dev_alloc_skb* 函数（在 Linux1.2 中没有）一个快捷方式，它用 GFP_ATOMIC 优先级调用 *alloc_skb*，并反转 skb->head 和 skb->data 之间的 16 个字节。这个数据空间可以用来“推”硬件包头。

void kfree_skb(struct sk_buff *skb, int rw);

void dev_kfree)skb(struct sk_buff *skb, int rw);

释放一个缓冲区。*kfree_skb* 被核心内部使用。驱动程序应该使用 *dev_kfree_skb*，在拥有缓冲区的套接字需要再次使用它的情况下，它可以正确地处理缓冲区加锁。两个函数的 rw 参数是 FREE_READ 或 FREE_WRITE。这个值用来跟踪套接字的内存。外出缓冲区应用 FREE_WRITE 来释放，而进来的则使用 FREE_READ。

unsigned char *skb_put(struct sk_buff *skb, int len);

这个线入函数更新结构 sk_buff 的 tail 和 len 域，它被用来在缓冲区尾加入数据。其返回值是 skb->tail 以前的值（或者说，它指向刚生成的数据空间）。有些驱动程序通过调用 ins(ioaddr,skb_put(…))或 memcpy(skb_put(…), data,len)来使用这个返回值。这个函数及下面的一些在为 Linux1.2 构造模块是不存在。

unsigned char *skb_push(struct sk_buff *skb, int len);

这个函数减小 skb->data，增加 skb->len。类似于 skb_put，除了数据是加在包开始而不是结尾。返回值指向刚生成的空间。

int skb_tailroom(struct sk_buff *skb);

这个函数返回为在缓冲区中放置数据的可用空间量。如果驱动程序在缓冲区中放了多于它能承载的数据，系统可能回崩溃。你也许会反对并认为，用 *printk* 指出这个错误已经足够了，而内存崩溃对系统太有害了，开发者肯定要采取一些措施。但实际上，如果缓冲区被正确分配了，你根本不必检查可用空间。因为驱动程序通常在分配缓冲区之前获

得包大小，只有有严重缺陷的驱动程序才可能在缓冲区内放太多的数据，崩溃可以认为是应得的惩罚。

int skb_headroom(struct sk_buff *skb);

返回数据前面得可用空间量，也就是可以向缓冲区中“推”多少八元组。

void skb_reserve(struct sk_buff *skb, int len);

这个函数增加 data 和 tail。它可以用来在填充缓冲区前预留空间。大多数以太网接口在包前预留两个字节；这样 IP 头可以在一个 4 字节以太网头之后，在 16 字节边界对齐。*snull* 完成得很好，尽管在“包接收”中并未提到这一点，那主要是为了避免彼时引入过多得概念。

unsigned char *skb)pull(struct sk_buff *skb, int len);

从包头中删除数据。驱动程序并不用这个函数，但为了完整性也包含在这里。它减少 skb->len，增加 skb->data；这是从进来包的开始剥出以太网包头的方法。

核心还定义了几个在套接字缓冲区上操作得别的函数，但它们主要应用于网络代码得高层，驱动程序并不需要它们。

地址解析

以太网通信最急迫得问题之一是硬件地址（接口得唯一标志符）与 IP 号码之间的关联。大多数协议都有类似问题，但我只向重点讨论一下以太网类得情况。我力图给出一个全面得描述，因此我将显示三种情况：ARP，没有 ARP 的以太网头(象 *plip*)，以及非以太网包头。

在以太网上使用 ARP

地址解析得一般方法是 ARP，即地址解析协议。幸运的是，ARP 由核心管理，以太网接口不必为支持 ARP 做任何特殊工作。只要在打开时正确地设置了 dev->addr 和 dev->addr_len，驱动程序不需担心任何从 IP 号码到物理地址的转换；*ether_setup* 将正确的设备方法赋给 dev->hard_header 和 dev->rebuild_header。

当一个包被构造时，以太网包头由 dev->hard_header 来布局，并由 dev->rebuild_header 在后来填充，它使用 ARP 协议将未知的 IP 号码映射到地址上。驱动程序作者不必知道这个过程的细节去写一个可工作得驱动程序。

越过 ARP

简单得点到点网络接口如 *plip* 可以从以太网包头获益，但却要避免来回发送 ARP 包得开销。*snull* 中得示例代码就属于这一类网络设备。*snull* 不能使用 ARP，因为驱动程序修改被发送得包得 IP 地址，而 ARP 包也交换 IP 地址。

如果你的设备想用一般的硬件包头，却不想运行 ARP，你需要越过缺省的 dev->rebuild_header 方法。这就是 *snull* 实现的方法，这个简单的函数有三条语句：

（代码 329）

事实上，并没有指定 eth->h_source 和 eth->h_dest 内容的实际需要，因为这些值只被用来进行包得物理传送，而一个点到点得连接保证能将包发送到它的目的地，而与硬件地址无关。*snull* 重构包头的原因是向你演示，当 *eth_rebuild_header* 不可用时，一个真实的网络接口的重构函数是如何实现的，

当接口收到一个包时，硬件包头只被 *eth_type_trans* 使用。我们在 *snull_rx* 中已经见过这个调用：

```
skb->protocol=eth_type_trans(skb,dev);
```

这个函数从以太网包头中抽取协议标志符（在这里是 ETH_P_IP）；它还要赋值 skb->mac.raw，从包数据中删去硬件包头，并设置 skb->pkt_type。最后一项在 skb 分配时缺省为 PACKET_HOST（表明包被指向这个主机），当然它也可以改为符合以太网目的地址得其它值。
如果你的接口是点到点连接，你将无法收到未想到的选播包。为避免这个，你必须记住那些第一个八元组的最低位（LSB）是 0 的目的地址将被指向单个主机（也就是说，它是 PACKET_HOST 或 PACKET_OTHERHOST）。*plip* 驱动程序用 0xfc 作为它的硬件地址的第一个八元组，而 *snull* 用 0x00。这两个地址都导致一个可工作的以太网类的点到点连接。

非以太网包头

这节简要地介绍硬件包头是如何用来封装相关信息的。如果你希望了解细节，你可以从核心源码或特别传输介质的技术文档中得到。我们刚才已经看到硬件包头除了含有目的地址外，还有一些信息，其中最重要的是通信协议。
不过，并不是每个协议都要提供所有的信息。象 *plip* 或 *snull* 之类的点到点连接可以避免传送整个以太网包头，同时不失去一般性。*hard_header* 设备方法从核心接收传送信息----包括协议级和硬件地址。它也收到 16 位的协议号。例如 IP 由 ETH_P_IP 标志。驱动程序应能正确地象接收主机传送包数据和协议号。点到点连接可以在硬件包头中省略地址，只传送协议号，因为传送是有保证的，与源和目的地址无关。一个只有 IP 的连接甚至什么硬件头都不传送。两种情况下，所有的工作都由 *hard_header* 完成，*rebuild_header* 除了返回 0 外什么都不做。
当包在连接的另一端被捡起时，接收函数将正确地设置 skb->protocol，skb->pkt_type，和 skb->mac.raw。
skb->mac.raw 是一个被网络高层代码实现的地址解析机制使用的字符指针（例如，net/ipv4/arp.c）。它必须指向一个与 dev->type 匹配的机器地址。设备类型的可能值被定义在 <linux/if_arp.h>；以太网接口用 ARPHRD_ETHER。例如，下面是 *eth_type_trans* 处理收到包的以太网包头的方法：

```
skb->mac.raw=skb->data;
skb_pull(skb, dev->hard_header_len);
```

在最简单的情况下（无包头的点到点连接），skb->mac.raw 可以指向一个含有这个接口的硬件地址的静态缓冲区，protocol 可以被置为 ETH_P_IP，packet_type 仍维持其缺省值 PACKET_HOST。

加载时配置

用户可以用几个标准的关键字来配置接口。任何新的网络模块都应遵循这个标准：

io=
: 为接口设置 I/O 端口的基地址。如果系统中安装了不只一个接口，那么可以用一个由逗号分隔的列表来指定。

irq=
: 设置中断号。和上面一样，可以指定不止一个值。

换句话说，一个装了两个 own_eth 接口的 Linux 用户可能用下面的命令行来加载模块：

```
insmod own_eth.o io=0x300, 0x320 irq=5,7
```

如果指定 0 值，那么 io=和 irq=选项都要被探测。因此用户可以通过指定 io=0 来强制探测。如果用户名优指定任何选项，多数驱动程序通常都探测一个接口，但有时，模块可能被禁止探测。（见 ne.c 中关于 NE2000 设备的探测）。
设备驱动程序应该象刚才描述的这样工作。ISA 设备的典型实现如下所示，假设驱动程序最多可以支持四个接口：
（代码 331）
这段代码缺省探测一个板子，并总是自动探测中断，但用户可以改变这种行为。例如，io=0,0,0 将探测三块板子。
除了使用 io 和 irq 外，驱动程序的作者可以随意增加其它加载时配置参数。也没有已建立的命名标准。

运行时配置

用户有时可能希望在运行时改变接口的配置。例如，当中断号无法探测时，想对它正确配置的唯一办法就是“ 尝试—错误 ”技术。一个用户空间的程序可以获取设备的当前配置，并通过在一个打开的套接字上调用 *ioctl* 来设置一个新的配置。例如，应用 *ifconfig* 使用 *ioctl* 为接口设置 I/O 端口。
我们前面知道一个为网络接口定义的方法中的一个是 *set_config*。这个方法被用来在运行时设置或改变一些接口特征。
当一个程序询问当前配置时，核心从结构 device 中抽取相关信息，而不通知驱动程序；另一方面，当一个新的配置被传递给接口时，*set_config* 被调用，这样驱动程序就可以检查这些值并采取相应的动作。这个驱动程序方法对应下面的原形：

```
int (*set_config)(struct device *dev, struct ifmap *map);
```

map 参数指向一个由用户程序传递的结构的拷贝；这个拷贝已经在核心空间，所以驱动程序不需要调用 *memcpy_from _fs*。
结构 ifmap 的域是：

```
unsigned long mem_start;
unsigned long mem_end;
unsigned short base_addr;
unsigned char irq;
unsigned char dma;
```

这些域对应着结构 device 中的域。

```
Unsigned char prot;
```

这个域对应着 dev 中的 if_port。map->port 的含义是设备特定的。

当一个进程为设备发出 ioctl(SIOCSIFMAP)(即 Socket I/O Control Set InterFace MAP)时，*set_config* 设备方法被调用。这个进程在时强制使用新值之前，应该发出 ioctl(SIOCGIFMAP)(即 Socket I/O Control Get InterFace MAP)，这样驱动程序只需要查看 dev 和 ifmap 结构不匹配的地方。Map 中任何不被驱动程序使用的域均可以略过。例如，一个不使用 DMA 的网络设备可以忽略 map->dma。
snull 实现被设计成可以显示驱动程序是如何针对配置改变而动作的。对 *snull* 来说，没有一个域由物理意义。但出于说明的目的，代码禁止改变 I/O 地址，允许改变 IRQ 号，并忽略其它选项，从而显示这些改变是如何被响应、拒绝、或是忽略的。

（代码 333 #1）
这个方法的返回值被作为发出的 *ioctl* 系统调用的返回值，对于没有实现 *set_config* 的驱动程序则返回-EOPNOTSUPP。
如果你对接口配置如何从用户空间访问感到好奇，请看 *misc-progs/netifconfig.c*，它可以用来与 *set_config* 比较。下面是一个示例运行的输出：
（代码 333 #2）

自定义 *ioctl* 命令

我们已经看到 *ioctl* 系统调用是为套接字实现的。SIOCSIFADDR 和 SIOCSIFMAP 是“套接字 *ioctl*”的例子现在让我们看看这个系统调用的第三个参数是如何被网络代码使用的。
当 *ioctl* 系统调用在套接字上被调用时，其命令号是在<linux/sockios.h>定义的符号之一，并且函数 *sock_ioctl* 直接调用一个协议特定的函数（这里协议指使用的主要网络协议，如 IP 或 AppleTalk）。
任何协议层不认识的 *ioctl* 命令被传递给设备层。这些设备相关的 *ioctl* 命令从用户空间接收第三个参数，即结构 ifreq*；这个结构在<linux/if.h>中定义。SIOCSIFADDR 和 SIOCSIFMAP 命令实际上是工作在 ifreq 结构上。SIOCSIFMAP 的额外参数，尽管定义为 ifmap，是 ifreq 的一个域。
除了使用标准的调用，每个接口可以定义它自己的 *ioctl* 命令。例如 *plip* 接口允许通过 *ioctl* 修改其内部超时值。套接字的 *ioctl* 实现将 16 个命令看作对接口是私有的：从 SIOCDEVPRIVATE 到 SIOCDEVPRIVATE+15。
当这些命令中的一个被认识时，dev->do_ioctl 在相关的接口驱动程序里被调用。这个函数接收与通用目的的 *ioctl* 函数使用的一样的 ifreq*指针。

```
Int (*do_ioctl)(struct device *dev, struct ifreq *ifr, int cmd);
```

Ifr 指针指向核心空间的一个地址，放有被用户传来结构的一个拷贝。在 *do_ioctl* 返回后，这个结构又被拷贝回用户空间；这样，驱动程序可以使用私有命令来接收和返回数据。
设备特定的命令可以选择使用结构 ifreq 中的域，但它们已经带有标准的含义，驱动程序不太可能根据自己的需要适配这个结构。域 ifr_data 是个 caddr_t 项（一个指针），用于设备特定的需要。驱动程序和调用 *ioctl* 命令的程序应在 ifr_data 的使用上取得一致。例如，*pppstats* 使用设备特定的命令来从 *ppp* 接口驱动程序中获取信息。
在这里不值得给出 *do_ioctl* 的一个实现，但根据本章的信息和核心的例子，你应能在需要的时候写出一个。不过注意，*plip* 实现不正确地使用了 ifr_data，不应做为 *ioctl* 实现的一个例子。

统计信息

一个驱动程序需要的最后一个方法是 *get_stats*。这个方法返回指向设备统计信息的一个指针。它的实现相当容易：
（代码 335）
返回有意义的统计信息所需的实际工作散布在驱动程序中，不同的域分别被更新。下表给出 enet_statistics 结构中最有趣的几个域。
int rx_packets;

int tx_packets;

这两个域含有接口成功传送的进来和外出包的总数。

int rx_errors;

int tx_errors;

出错的接收和发送的数目。接收错可能是错误的校验和、错误的包大小，以及其它问题的结果。发送错误不太常见，一般都是线缆的问题。

int rx_dropped;

int tx_dropped;

在接收和发送时被丢弃的包的个数。当包数据没有可用内存时，包便被丢弃了。tx_dropped 很少使用。

这个结构还有几个域，可以用来细分发送和接收时发生的错误。感兴趣的读者可以看<linux/if_ether.h>中结构的定义。

选播

“选播”包是指一个网络包，它将要被多于一个，但又不是全部的主机接收。

这个功能是通过给一组主机赋以特殊的硬件地址来获得的。指向这些特殊地址中的一个的包将被这个组中所有的主机收到。在以太网的情况下，一个选播地址是将目的地址中第一个八元组的最低位设置而得到，而所有的设备板子都在其硬件地址中将这一位清除。

处理主机组以及硬件地址的棘手部分都有应用或核心完成了，接口驱动程序并不需要处理这些问题。

选播包的发送很简单，与其它包完全一样。接口在传输介质上发送它们，根本不管目的地址。核心必须设置一个正确的硬件目的地址；*rebuild_header* 设备方法（如果被定义）并不需要查看它整理的数据。

而另一方面，接收选播包需要设备的一些合作。当一个“有趣的”选播包被收到时（也就是一个包的目的地址确定一组主机，其中包含了这个接口），硬件应该通知操作系统。这意味着硬件过滤器应被设计为能够区别不同的选播地址。这个过滤器在接口的一般操作中，将网络包的地址与其自己的硬件地址进行匹配。

典型地，在考虑选播的情况下，硬件可分为一下三类：

- 不能处理选播的接口。这类接口要么接收指向自己硬件地址的包（包括播送包），要么节收所有的包。它们接收选播包是通过接收所有的包实现的，这样，操作系统中会充斥着大量“无意义”的包。一般我们不认为这类接口为支持选播的，驱动程序不在 dev->flags 中置 IFF_MULTICAST。

  点到点接口是一种特殊情况，它们通常接收所有的包，根本不进行任何硬件过滤。
- 能区别选播包和其它包（主机到主机或播送）的接口。这类接口可以被要求接收所有的选播包，然后用软件来判断自己是否是有效的接收者。这种情形引入的开销是可以接收的，因为一个典型的网络中选播包的数量都很少。
- 能够进行选播地址硬件检测的接口。可以给这类接口一组需要接收的选播地址，它们会忽略其它的选播包。这对核心来说是最优化的情况，因为不会浪费处理器事件去丢弃接口收到的“无意义”的包。

核心试图利用高级接口的能力，能最好地支持第三类接口（用途最广）。因此，当有效的选播地址被改变时核心应通知驱动程序，它把新的一组地址传给驱动程序，这样它可以按照新的信息更新硬件过滤器。

核心对选播的支持

下面是与驱动程序的选播能力相关的数据结构和函数的概括：

void (*dev->set_multicast_list)(struct device *dev)

当与设备相关的机器地址表改变时调用这个设备方法。当 dev->flags 被修改时它也被调用，因为有些标志也要求你重新配置硬件过滤器。这个方法接收一个指向 device 结构的指针作为参数，返回 void。对实现这个方法不感兴趣的驱动程序可以留域为 NULL。

struct dev_mc_list *dev->mc_list

这是域设备相关的所有选播地址的链表。这个结构的实际定义在本节结束时介绍。

int dev->mc_count

链表项数。这个信息有点冗余，但检查 mc_count 是否为 0 是优于列表检查的一个有用的快捷方式。

IFF_MULTICAST

除非驱动程序在 dev->flags 中设置了这个标志，接口将不必处理选播包。当 dev->flags 改变时，至少 set_multicast_list 会被调用。

IFF_ALLMULTI

dev->flags 中的这个标志被网络软件设置以告诉驱动程序从网络中抽取所有播送包。这在 multicast-routing 被使用时发生。如果这个标志被置位，dev->mc_list 将不再使用去过滤选播包。

IFF_PROMISC

当接口被置为杂乱模式时，dev->flags 中的这个标志被设置。所有的包都被接口抽取，不考虑 dev->mc_list。

驱动程序开发者需要的最后一点信息是结构 dev_mc_list 的定义，它居于<linux/netdevice.h>中。

（代码 337）

由于选播和硬件地址与包的实际发送无关，这个结构可在不同的网络实现上移植，每个地址由一串八元组和一个长度确定，就象 dev->dev_addr。

一个典型的实现

描述 *set_multicast_list* 设计的最号办法是给出一些伪代码。

下面的函数是这个函数在一个全特征(ff)驱动程序中的一个典型实现。说这个驱动程序是全特征的是因为它控制的接口有一个复杂的硬件包过滤器，它可以存放一个由本机接收的选播地址表。这个表的最大尺寸是 FF_TABLE_SIZE。

所由带有前缀 ff_的函数是放置硬件特定操作的地方。

（代码 338）

如果这个接口不能在硬件过滤器中存储到来包的选播表，那么这个实现还可以简化。在这种情况下，FF_TABLE_SIZE 减为 0，代码的最后四行也不需要了。

现在，接口板一般不能存储选播表。不过，这并不是一个大问题，因为网络代码的高层会负责将不需要的包丢弃。
如我前面建议的，即使不能处理选播包的接口也需要实现 *set_multicast_list* 方法，这样当 dev->flags 发生改变时可以被通知。我称这个为“ 无特征 ”(nf)的实现。它非常简单，如下面所示：
（代码 339）
处理 IFF_PROMISC 是很重要的，因为不然的话，用户将无法运行 *tcpdump* 或其它一些网络分析工具。另一方面，如果接口运行一个点到点的连接，则没有任何实现 *set_multicast_list* 的必要，因为它们接收所有的包。

快速参考

本节提供在本章中介绍的概念的参考。它也解释了驱动程序应包含的每个头文件的作用。不过，device 和 sk_buff 的每个域的列表，就不再重复了。

#include <linux/netdevice.h>

这个头文件含有 struct device 的定义，还包含了网络驱动程序需要的几个其它头文件。

void netif_rx(struct sk_buff *skb);

这个函数在中断时可以被调用通知核心一个包被收到了，并且封装在一个套接字缓冲区中。

#include <linux/if.h>

被 *netdevice.h* 包含，这个文件声明接口标志(IFF_macros)和结构 ifmap，它在网络驱动程序的 *ioctl* 实现中用重要的作用。

#include <linux/if_ether.h>

ETH_ALEN

ETH_P_IP

struct ethhdr;

struct enet_statistics;

被 *netdevice.h* 包含，*if_ether.h* 定义所有的 ETH_macros，用来表示八元组长度（象地址长度）和网络协议（象 IP）。它也定义了结构 ethhdr 和 enet_statistics。注意，不要看 enet_statistics 的名字和包含它的头文件，事实上，所有的接口都要用到它，不仅仅是以太网。

#include <linux/skbuff.h>

结构 sk_buff 和一些相关结构的定义，以及在缓冲区上操作的线入函数。这个头文件包含在 *netdevice.h* 中。

#include <linux/etherdevice.h>

void ether_setup(struct device *dev)；

这个函数为以太网驱动程序设置大多数设备方法为通用目的的实现。它同样设置 dev->flags，并且在名字中的第一个字符是空格或空字符时，将下一个可用的 ethx 名赋给 dev->name。

unsigned short eth_type_trans(struct sk_buff *skb, struct device *dev)

当以太网接口收到一个包，这个函数将被调用来设置 skb->pkt_type。返回值是一个协议号，通常被存在 skb->protocol 中。

#include <linux/sockios.h>

SIOCDEVPRIVATE

这是 16 个 *ioctl* 命令的第一个，可以被每个驱动程序实现以供私用。所有的网络 *ioctl* 命令都在 *sockios.h* 中定义。

第十五章　外围总线概览

在第八章“硬件管理”中，我们介绍了最低级的硬件控制，本章提供一个较高级的总线体系结构的概览。总线由电气接口和编程接口组成。在这一章，我打算介绍编程接口。
本章覆盖了几种总线体系结构。不过，基本重点是访问 PCI 外围的核心功能，因为近来，PCI 总线是最常用的外围总线，也是核心支持最好的总线。

PCI 接口

尽管很多计算机用户认为 PCI（外围部件互连，Peripheral Component Interconnect）是布局电气线路的一种方法，但实际上，它是一组完全的规范，定义了计算机的不同部分是如何交互的。
PCI 规范覆盖了与计算机接口相关的绝大多数方面。我不打算在这里全部介绍，在本节中，我主要关心一个 PCI 驱动程序是如何找到它的硬件，并获得对它的访问的。在第二章“构造和运行模块”的“自动和手工配置”一节，及在第九章“中断处理”的“自动检测中断号”一节中讨论过的探测技术同样可以应用于 PCI 设备，但规范还提供了探测的另外办法。
PCI 结构被设计来替代 ISA 标准，由三个主要目标：在计算机和其外围之间传送数据时有更高的性能，尽可能地做到平台无关性，使在系统中增减外围设备得到简化。
PCI 通过使用比 ISA 高的时钟频率来获得更高的性能；它的时钟运行在 25 或 33MHZ（实际时钟是系统时钟的几分之一的整数倍），而且马上就会游 66MHZ 的扩展。另外，它被装配在 32 位的数据总线上，64 位的扩展正在规范中。平台无关性一直是计算机总线的一个设计目标，这是 PCI 的尤其重要的一个特征，因为 PC 世界一直以来总是被处理器特定的标准所主宰。
不过对驱动程序作者来说，最要紧的是对接口板自动检测的支持。PCI 设备是无跳线的（与大多数 ISA 外围不同），并且在引导时被自动配置。因此，设备驱动程序必须能访问设备上的配置信息来完成初始化。这些情形都不需要任何探测。

PCI 寻址

每个外围由一个总线号、一个设备号、和一个功能号确定。虽然 PCI 规范允许一个系统最多拥有 256 条总线，但 PC 只有一条。每条总线最多带 32 个设备，但每个设备可以是最多个功能的多功能板（如一个音频设备带一个 CD-ROM 驱动器）。每个功能可以由一个 16 位的键或两个 8 位的键确定。Linux 核心采用后一种方法。
每个外围板子的硬件电路回答与三个地址空间相关的询问：内存位置，I/O端口，和配置寄存器。前两个地址空间由PCI总线上的所有设备共享（也就是说，当你访问一个内存位置，所有的设备都将同时看到这个总线周期）。而配置空间则利用“地理寻址”，每个槽有一个配置事务的私用使能线，PCI控制器一次访问一个板子，不会有地址冲突。考虑到驱动程序，内存和I/O是以通常的*inb*,*memcpy*等来访问。而配置事务则通过调用特定的核心函数访问配置寄存器来完成。至于中断，每个PCI设备有 4 个中断管脚，它们到处理器中断线的路由是主板的任务；PCI中断可以设计为共享的，这样即使是一个有限中断线的处理器也能带很多PCI接口板。
PCI 总线的 I/O 空间使用 32 位的地址总线（这样就是 4GB 的 I/O 端口），而内存空间则可以用 32 位或 64 位地址访问。地址对每个设备来说应该是唯一的，但也有可能有两个设备错误

地映射到同一个地址，使得哪个都不能被访问。一个好消息是接口卡提供的每个内存和 I/O 地址区段都可以通过配置事务重映射。这就是设备可以在引导时被初始化从而避免地址冲突的机制这些区段当前映射到的地址可以从配置空间读出，因此 Linux 驱动程序可以不通过探测就访问其设备。一旦配置寄存器被读出，驱动程序就可以安全的访问它的硬件。

PCI配置空间由每个设备函数 256 个字节构成，配置寄存器的布局是标准化的。配置空间有四个字节含有一个唯一的函数ID，因此驱动程序可以通过在外围查找特定的ID来B确定它的设备*。总之，每个设备板子被地理寻址以取得它的配置寄存器；这个信息可以用来确定这个板子或采取进一步动作。

从前面的描述，应该清楚 PCI 接口标准比 ISA 的主要创新是配置地址空间。因此，除了通常的驱动程序代码外，PCI 驱动程序还需要访问配置空间的能力。

在本章的其余部分，我将使用单词“ 设备 ”来指一个设备功能，因为多功能板上的每个功能均是一个独立的实体。当我提到一个设备，我是指元组“ 总线号，设备号，功能号 ”。如前所述，每个元组在 Linux 中由两个 8 位数字表示。

引导时

让我们看一下 PCI 是如何工作的，从系统引导开始，因为那时设备被配置。

当 PCI 设备被加电时，硬件关闭。或者说，设备只对配置事务响应。加电时，设备没有映射到计算机地址空间的内存和 I/O 端口；所有其它的设备特定的特征，象中断线，也都被关闭。

幸运的是，每个 PCI 母板都装有懂得 PCI 的固件，根据平台的不同被称做 BIOS、NVRAM、或 PROM。固件提供对设备配置地址空间的访问，即使处理器的指令集不提供这样的能力。在系统引导时，固件对每个 PCI 外围执行配置事务，从而为它提供的任何地址区段分配一个安全的地方。到设备驱动程序访问设备时，它的内存和 I/O 区段已经被映射到处理器的地址空间。驱动程序可以改变这个缺省的分配，但它通常并不这样做，除非有一些设备相关的原因要求这样。

在 Linux 中，用户可以通过读*/proc/pci* 来查看 PCI 设备，这是个文本文件，系统中每个 PCI 板子有一项。下面是*/proc/pci* 中一项的例子：

（代码 344）

/proc/pci 中每一项是一个设备的设备无关特征的概述，如它的配置寄存器所描述的。例如，上面这一项告诉我们这个设备有板上内存，已被映射到地址 0xf1000000。一些古怪的细节的含义以后在我介绍过配置寄存器后将会清楚。

检测设备

如前面提到的，配置空间的布局是设备无关的。在这一节，我们将看看用来确定外围的配置寄存器。

PCI 设备有一个 256 字节的地址空间。前 64 个字节是标准化的，而其余的则是设备相关的。图 15-1 显示了设备无关配置空间的布局。

如图所示，有些 PCI 的配置寄存器是要求的，而有些则是可选的。每个 PCI 设备必须在必要寄存器中包含有意义的值，而可选寄存器的内容则以来与实际外围的能力。可选域并不使用，除非必要域的内容表明它们是有效的。这样，必要域断言了板子的能力，包括其它域可用与否。

有意思的是注意到 PCI 寄存器总是小印地安字节顺序的。尽管标准要设计为体系结构无关的，PCI 的设计者有时还是显示出对 PC 环境的偏见。驱动程序的作者应该留神字节顺序，

特别是访问多字节的配置寄存器时；在 PC 上工作的代码可能在别的平台上就不行。Linux 的开发者已经注意了字节排序问题（见下一节“访问配置空间”），但这个问题还是要牢记在心。不幸的是，标准函数 *ntohs* 和 *ntohl* 都不能用，因为网络字节顺序与 PCI 顺序相反；在 Linux2.0 中没有标准函数将 PCI 字节顺序转换为主机字节顺序，每个用单个字节构成多字节值的驱动程序都应该特别小心地正确处理印地安字节序。核心版本 2.1.10 引入了几个函数来处理这些字节顺序问题，它们在第十七章“最近的发展”中“转换函数”一节介绍。

（图 15-1：标准化的 PCI 配置寄存器）

描述所有的配置项超出了本书的范围。通常，与设备一起发布的技术文档会描述它支持的寄存器。我们感兴趣的是驱动程序如何找到它的设备，以及它如何访问设备的配置空间。

三个 PCI 寄存器确定一个设备：销售商，设备 ID，和类。每个 PCI 外围把它自己的值放入这些只读寄存器，驱动程序可以用它们来查找设备。让我们更仔细地看看这些寄存器：

销售商

这个 16 位的寄存器确定硬件的生产商。例如，每个 Intel 的设备都会标上同样的销售商号，8086 hex（是个随即值？）。这样的号码有一个全球的注册，生产商必须申请一个唯一的号。

设备 ID

这是另一个 16 位寄存器，由生产商选择；不需要有官方的注册。这个 ID 通常与销售商 ID 成对出现，形成一个硬件设备的唯一的 32 位标志符。我将用单词“签名”来指销售商/设备 ID 对。一个设备驱动程序经常以来于签名来确定它的设备；驱动程序的作者从硬件文档中知道要寻找什么值。

类

每个外围设备都属于一个类。类寄存器是个 16 位的值，它的高八位确定“基类”（或组）。例如，“以太网”和“令牌环”是属于“网络”组的两类，而“串行”和“并行”类属于“通信”组。有些驱动程序可以支持几种类似的设备，它们虽然有不同的签名，却属于同一类；这些驱动程序可以依赖于类寄存器来确定它们的外围，如以后所示。

下面的头文件，宏，以及函数都将被 PCI 驱动程序用来寻找它的硬件设备：

#include <linux/config.h>

驱动程序需要知道是否 PCI 函数在核心是可用的。通过包含这个头文件，驱动程序获得了对 CONFIG_宏的访问，包括 CONFIG_PCI(将在下面介绍)。从 1.3.73 以来，这个头文件包含在<linux/fs.h>中；如果想向后兼容，你必须把它显式地包含。

CONFIG_PCI

如果核心包括对 PCI BIOS 调用的支持，那么这个宏被定义。并不是每个计算机都有 PCI 总线，所以核心的开发者应该把 PCI 的支持做成编译时选项，从而在无 PCI 的计算机上运行 Linux 时节省内存。如果 CONFIG_PCI 没有定义，那么这个列表中其它的函数都不可用，驱动程序应使用预编译的条件语句将 PCI 支持全都排除在外，以避免加载时的“未定义符号”错。

#include <linux/bios32.h>

这个头文件声明了本节介绍的所有的原型，因此一定要被包含。这个头文件还定义了函数返回的错误代码的符号值。它在 1.2 和 2.0 之间没有改变，因此没有可移植性问题。

int pcibios_present(void)

由于 PCI 相关的函数在无 PCI 的计算机上毫无意义，*pcibios_present* 函数就是告诉驱动程序计算机是否支持 PCI；如果 BIOS 懂得 PCI，它返回一个为真布尔值。即使

---

* 你可以在它自己的硬件手册中找到任何设备的ID。

CONFIG_PCI 被定义了，PCI 功能仍是一个运行时选项。因此，你在调用下面介绍的函数之前要检查一下 *pcibios_present*，保证计算机支持 PCI。

```
#include <linux/pci.h>
```

这个头文件定义了下面函数使用的所有数值的符号名。并不是所有的设备 ID 都在这个文件中列出了，但你在为你的 ID，销售商，类定义宏之前，最好还是看看这个文件。注意这个文件一直在变大，因为不断有新设备的符号定义被加入。

```
int pcibios_find_device(unsigned short vendor, unsigned short id, unsigned short index,
                        unsigned char *bus, unsigned char *function);
```

如果 CONFIG_PCI 被定义了，并且 *pcibios_present* 也是真，这个函数被用来从 BIOS 请求关于设备的信息。销售商/ID 对确定设备。index 用来支持具有同样的销售商/ID 对的几个设备，下面将会解释。对这个函数的调用返回设备在总线上的位置以及函数指针。返回代码为 0 表示成功，非 0 表示失败。

```
int pcibios_find_class(unsigned int class_code, unsigned short index,
                       unsigned char *bus, unsigned char *function);
```

这个函数和上一个类似，但它寻找属于特定类的设备。参数 class_code 传递的形式为：16 位的类寄存器左移八位，这与 BIOS 接口使用类寄存器的方式有关。这次还是，返回代码为 0 表示成功，非 0 表示有错。

```
char *pcibios_strerror(int error);
```

这个函数用来翻译一个 PCI 错误代码（象 *pcibios_find_device* 返回的）为一个字符串。你也许在查找函数返回的即不是 PCIBIOS_SUCCESSFUL(0)，也不是 PCIBIOS_DEVICE_NOT_FOUND 时（这是当所有的设备都被找过以后所期望返回的错误代码），希望打印一条错误信息。

下面的代码是驱动程序在加载时检测设备所使用的典型代码。如上面所提到的，这个查找可以基于签名或者设备类。不管是哪种情况，驱动程序不许存储 bus 和 function 值，它们在后面确定设备时要用到。function 的前五位确定设备，后三位确定函数。

下面的代码中，每个设备特定的符号加前缀 jail_（另一个指令列表），大写或小写依赖于符号的种类。

如果驱动程序可以依赖于唯一的销售商/ID 对，下面的循环可以用来初始化驱动程序：

（代码 347）

（代码 348）

如果这个代码段只处理由 JAIL_VENDOR 和 JAIL_ID 确定的一类 PCI 设备，那么它是正确的。

不过，很多驱动程序非常灵活，能够同时处理 PCI 和 ISA 板子。在这种情况下，驱动程序仅在没有检测到 PCI 板子或 CONFIG_PCIBIOS 没有定义时才探测 ISA 设备。

使用 *pcibios_find_class* 要求 *jail_init_dev* 完成比例子中要多的工作。只要它找到了一个属于指定类的设备，这个函数就成功返回，但驱动程序还要确认其签名也是被支持的。这个任务通过一系列的条件语句完成，结果是抛弃很多不期望的设备。

有些 PCI 外围包含通用目的的 PCI 接口芯片和设备特定的电路。所有使用同样接口芯片的外围板子都有同样的签名，驱动程序必须进行额外的探测以保证它在处理正确的外围设备。因此，有时象 *jail_init_dev* 之类的函数必须准备好做一些设备特定的额外的检测，以抛弃那些可能有正确签名的设备。

访问配置空间

在驱动程序检测到设备后，它通常要对三个地址空间读或写：内存、端口和配置。特别地，访问配置空间对驱动程序来说极为重要，以呢这是它发现设备被映射到内存和 I/O 空间什么地方的唯一的办法。
由于微处理器无法直接访问配置空间，计算机销售商必须提供一个办法来完成它。准确的实现因此是销售商相关的，与我们这里的讨论无关。幸运的是，这个事务的软件接口（下面描述）是标准化的，驱动程序或 Linux 核心都不需要知道它的细节。
至于驱动程序，配置空间可以通过 8 位、16 位、32 位的数据传送来访问。相关函数的原型在<linux/bios32.h>：

```
int pcibios_read_config_byte(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_word(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_dword(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
```

从由 bus 和 function 确定的设备的配置空间读取 1，2，4 个字节。参数 where 是从配置空间开始处的字节偏移。 从配置空间取出的值通过 ptr 返回，这些函数的返回值是错误代码。字和双字函数将刚从小印地安字节序读出的值转换为处理器本身的字节序，因此你并不需要处理字节序。

```
int pcibios_write_config_byte(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned char val);
int pcibios_write_config_word(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned short val);
int pcibios_write_config_dword(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned int val);
```

向配置空间里写 1，2，4 个字节。设备仍由 bus 和 function 确定，要写的值由 val 传递。字和双字函数在向外围设备写之前将数值转换为小印地安字节序。
访问配置变量的最好办法是使用在<linux/pci.h>中定义的符号名。例如，下面的两行程序通过给 *pcibios_read_config_byte* 的 where 传递符号名来获取一个设备的修正 ID。

```
Unsigned char jail_get_revision(unsigned char bus, unsigned char fn)
{
    unsigned char *revision;

    pcibios_read_config_byte(bus,fn, PCI_REVISION_ID,&revision);
    return revision;
}
```

当访问多字节值时，程序远一定要记住字节序的问题。

看看一个配置快照

如果你向浏览你系统上 PCI 设备的配置空间，你可以编译并加载模块 *pci/pcidata.c*，它在 O’Reilly FTP 站点上提供的源文件中。
这个模块生成一个动态的*/proc/pcidata* 文件，包含有你的 PCI 设备配置空间的二进制快照。这个快照在文件每次被读时更新。*/proc/pcidata* 的大小被限制为 PAGE_SIZE 字节(这是动态 */proc* 文件的限制，在第四章“ 调试技术 ”中“ 使用*/proc* 文件系统 ”一节介绍过)。这样，它只列出前 PAGESIZE/256 个设备的配置内存，意味着 16 或 32 个设备（也许对你的系统已经

够了）。我选择把*/proc/pcidata* 作成二进制文件，而不是象其它*/proc* 文件那样是文本的，就是因为这个大小限制。

pcidata 的另一个限制是它只扫描系统的第一条 PCI 总线。如果你的系统有到其它 PCI 总线的桥，*pcidata* 将忽略它们。

在*/proc/pcidata* 中设备出现的顺序与*/proc/pci* 中相反。这是因为*/proc/pci* 读的是一个从头部生长的链表，而*/proc/pcidata* 则是一个简单的查找循环，它按照取到的顺序将所有的东西输出。

例如，我的抓图器在*/proc/pcidata* 的第二个出现，（目前）有下面的配置寄存器：

```
morgana% dd bs=256 skip=1 count=1 if=/proc/pcidata | od –Ax –t x1
1+0 records in
1+0 records out
000000 86 80 23 12 06 00 00 02 00 00 00 04 00 20 00 00
000010 00 00 00 f1 00 00 00 00 00 00 00 00 00 00 00 00
000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000030 00 00 00 00 00 00 00 00 00 00 00 00 0a 01 00 00
000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

如果你将上面的输出和图 15-1 比较，你就可以理解这些数字。或者，你可以使用 *pcidump* 程序，在可以从 FTP 站点上找到，它将输出列表格式化并标号。

pcidump 的代码并不值得在这儿列出，因为这个简单程序只是一个长表，外加十行扫描这个表的代码。相反，让我们看看一些选择的输出行：

（代码 351）

pcidata 和 *pcidump*，与 *grep* 配合使用，对调试驱动程序的初始化代码非常有用。不过注意，*pcidata.c* 模块是 GPL 的，因为我是从核心源码中取的 PCI 扫描循环。这不应该对你作为一个驱动程序的作者有什么影响，因为我只是以一个支持工具的形式将这个模块包含在源文件中，而不是新驱动程序的可重用模版。

访问 I/O 和内存空间

一个PCI外围实现六个地址区段。每个区段由内存或I/O位置组成，或者压根不存在。大多数设备用一个内存区段代替它们的I/O端口，因为有些处理器（象Alpha）没有本身的I/O空间，还因为PC上的I/O空间都相当拥挤。内存和I/O空间的结构化的不同通过实现一个“ 内存可预取 ”位*来表达。将其控制寄存器映射到内存地址范围的外围将这个范围声明为不可预取的，而PCI板子上的有些东西如视频内存是可预取的。在本节中，只要讨论适用于内存或I/O，我就用单词“ 区段 ”来指一个PCI地址范围。

一个接口板子用配置寄存器（在图 15-1 中所示的 6 个 32 位寄存器，它们的符号名从 PCI_BASE_ADDRESS_0 到 PCI_BASE_ADDRESS_5）报告它的区段的大小和当前位置。由于 PCI 定义的 I/O 空间是一个 32 位的地址空间，因此用对内存和 I/O 适用同样的配置接口是可行的。如果设备使用 64 位的地址总线，它可以为每个区段用两个连续的 PCI_BASE_ADDRESS 寄存器在 64 位的内存空间来声明区段。因此有可能一个设备同时提供 32 位和 64 位的区段。

我不想在这儿讨论太多的细节，因为如果你打算写一个 PCI 驱动程序，你总会这个设备的硬件手册的。特别地，我不打算使用寄存器的预取位或两个“ 类型 ”位，并且我将讨论限制在 32 位外围上。不过，了解一下一般情况下是如何实现的，以及 Linux 驱动程序是如何处理 PCI 内存是很有趣的。

PCI 规范要求每个被实现的区段百升微被映射到一个可配置地址上。这意味着设备必须位它实现的每个区段装备一个可编程 32 位解码器，并且利用 64 位 PCI 扩展的板子必须有一个 4 位可编程解码器。尽管在 PC 上没有 64 位 PCI 总线，一些 Alpha 工作站则有。

由于通常一个区段的字节数是 2 的幂，如 32、64、4KB 或 2MB，所以可编程解码器的实际实现和使用都被简化了。而且，将一个区段映射到一个未对齐的地址上意义也不大；1MB 的区段自然在 1M 整数倍的地址处对齐，32 字节的区段则在 32 的整数倍处。PCI 规范利用了这个对齐 ;它要求地址解码器需要且只需查看地址总线的高位,并且只有高位是可编程的。这个约定也意味着任何区段的大小都必须是 2 的幂。

这样，重映射一个 PCI 区段可以通过在配置寄存器的高位设置一个合适的值来完成。例如，一个 1M 的区段，有 20 位的地址空间，可以通过设置寄存器的高 12 位进行重映射；向寄存器写 0x008xxxxx 告诉板子对 8MB-9MB 的地址区间响应。实际上，只有非常高的地址被用来映射 PCI 区段。

这种“ 部分解码 ”有几个额外的好处就是软件可以通过检查配置寄存器中非可编程位的数目来确定 PCI 区段的大小。为了这个目的，PCI 标准要求未使用的位必须总是读作 0。通过强制 I/O 区段的最小大小为 8 字节，内存区段为 16 字节，标准可以把一些额外的信息放入同一个 PCI 寄存器中：“ 空间 ”位，表明区段是内存的还是 I/O 的；两个“ 类型 ”位；一个“ 预取 ”位，只是位内存定义的。类型位在 32 位区段、64 位区段、以及 “ 必须映射在 1M 一下的 32 位区段 ” 进行选择。最后这个值用于那些仍然运行于一些 PC 上的过时软件。

检测一个 PCI 区段的大小可以通过使用几个定义在<linux/pci.h>中的位掩码来简化：是个内存区段时 PCI_BASE_ADDRESS_SPACE 被置位 ;PCI_BASE_ADDRESS_MEM_MASK 为内存区段掩去配置位 ;PCI_BASE_ADDRESS_TO_MASK 位 I/O 区段掩去这些位。规范还要求地址区段必须按序分配，从 PCI_BASE_ADDRESS_0 到 PCI_BASE_ADDRESS_5；这样一旦一个基地址未用（也就是被置未 0），你就可以知道所有的后续地址都未用。

报告 PCI 区段当前位置和大小的典型代码如下：

（代码 353）

（代码 354 #1）

这个代码是 *pciregion* 模块的一部分，与 *pcidata* 在同一个目录下发布；这个模块生成一个 */pci/pciregions* 文件，用上面给出的代码产生数据。当配置寄存器被修改时，中断报告被关闭，以防止驱动程序访问被映射到错误位置的区段。使用 *cli* 而不是 *save_flags* 是因为这个函数只在 *read* 系统调用时被执行，我们知道在系统调用的时候中断是打开的。

例如，这里是我的抓图器的*/proc/pciregion* 的报告：

（代码 #2）

计算机的固件在引导时用一个类似于前面给出的循环来正确地映射区段。由于固件防止了任何地址赋值时的冲突，Linux 驱动程序通常并不改变 PCI 区间的映射。

有趣的是注意到上面的程序报告的内存大小有可能被夸大。例如，*/proc/pciregion* 报告说我的视频板子是一个 16MB 的设备。但这并不真实（尽管我有可能扩展我的视频 RAM）。但由于这个大小信息只是被固件用来分配地址区间,夸大区段大小对驱动程序的作者来说并不是一个问题，他设备的内部并能正确地处理由固件分配的地址区间。

PCI 中断

至于中断，PCI 很容易处理。计算机的固件已经给设备分配了一个唯一的中断号，驱动程序只需要去用它即可。中断号存在配置寄存器 60 中（PCI_INTERRUPT_LINE），它是一个字节宽。这允许最多 256 条中断线，但实际限制依赖于使用的 CPU。驱动程序不必麻烦去检

---

* 这个信息居于基地址PCI寄存器的某个低序位中。

查中断号，因为在 PCI_INTERRUPT_LINE 中找到的一定是正确的。
如果设备不支持中断，寄存器 61（PCI_INTERRUPT_PIN）为 0；不然为非 0。不过由于驱动程序知道它的设备是否是中断驱动的，因此并不常需要去读 PCI_INTERRUPT_PIN。
这样，处理中断的 PCI 特定的代码只需要这个配置字节以取得中断号，如下面所示的代码。不然，应用第九章的信息。

```
result = pcibios_read_config_byte(bus,fnct,PCI_INTERRUPT_LINE, &my_irq);
if(result){/*deal with result*/}
```

本节的其余部分为感兴趣的读者提供一些额外的信息，但对写驱动程序并不需要。
一个 PCI 连接器有四个中断脚，外围板子可以任意使用。每个管脚都是独立地路由到主板的中断控制器，因此中断可以共享，而没有任何电气问题。中断控制器负责将中断线（脚）映射到处理器的硬件；将这个平台相关的操作留给控制器是为了获得总线本身的平台无关性。
位于 PCI_INTERRUPUT_PIN 的只读配置寄存器用来告诉计算机哪一个管脚被使用了。值得记住的是每个设备板子最多可带 8 个设备；每个设备使用一个中断脚并在它自己的配置寄存器中报告它。同一个设备板子的不同设备可以使用不同的中断脚，也可以共享同一个。
另一方面，PCI_INTERRUPT_LINE 寄存器是读/写的。在计算机引导时，固件扫描它的 PCI 设备，并按照中断脚是如何路由到它的 PCI 槽的为每个设备设置这个寄存器。这个值由固件来赋，因为只有固件知道母板是如何将不同的中断脚路由到处理器的。然而，对设备驱动程序来说，PCI_INTERRUPT_LINE 寄存器是只读的。

回顾：ISA

ISA 总线在设计上相当老了，而且在性能方面也名声扫地，但它依然占据着扩展设备的很大一块市场。如果速度不是很重要，并且你想支持旧的主板，那么 ISA 实现要比 PCI 更令人喜欢。这个旧标准的一个额外的优势是，如果你是个电子爱好者，你可以很容易地构造你自己的设备。
令一方面，ISA 的一个巨大的缺点是它紧密地绑定在 PC 体系结构上；接口总线具有 80286 处理器的所有限制，导致系统程序员无穷的痛苦。ISA 设计的令一个巨大的问题（从原先的 IBM PC 继承下来的）是缺乏地理寻址，这导致了无穷的问题和为加一个新设备时漫长的“ 拔下--重跳线—插上—测试 ”循环。有趣的是注意到即使是最老的 Apple II 计算机都已经利用了地理寻址，它们的特征是无跳线的扩展板。

硬件资源

一个 ISA 设备可以装配 I/O 端口，内存区域，和中断线。
即使 x86 处理器支持 64KB 的 I/O 端口内存（也就是说，处理器申明 16 根地址线），有些老的 PC 硬件也只能对最低的 10 根地址线解码。这将可用的地址空间限制为 1024 个端口，因为在 1KB-64KB 区间的任何地址会被任何只能解码低地址线的设备错误地看成低地址。一些外围通过只映射一个端口到低 KB，并使用高地址线在不同的设备寄存器中选择的办法绕过了这个限制。例如，一个映射到 0x340 的设备可以安全地使用端口 0x740，0xB40，等等。如果说 I/O 端口的可用性受到了限制，那么内存访问就更糟了。一个 ISA 设备只能使用 640KB-1MB 和 15MB-16MB 之间的内存区间。640KB-1MB 区间被 PC BIOS、VGA 兼容的视频板、以及各种其它设备使用，留给新设备很少的可用空间。另一方面，15M 处的内存，Linux 并不直接支持；这个问题在第八章的“ 访问设备板子上的内存 ”讨论过。

ISA 设备板子上第三个可用的资源是中断线。有限的中断线被路由到 ISA 总线，它们被所有的接口板共享。造成的结果是，如果设备没有被正确地配置，它们可以用同样的中断线找到它们自己。
尽管原先的ISA规范不允许跨设备的中断共享，多数设备板子还是允许的*。软件级的中断共享在第九章的“ 中断共享 ”中描述过。

ISA 程序设计

至于程序设计，除了 Linux 核心通过维护 I/O 和 IRQ 寄存器提供有限的帮助外（在第二章中的“ 使用资源 ”和第九章的“ 安装中断处理程序 ”中描述过），核心和 BIOS 中没有任何东西使得使用 ISA 设备更容易一些。
本书整个第一部分给出的编程技巧同样适用于 ISA 设备；驱动程序可以探测 I/O 端口，中断线必须用在第九章“ 自动检测 IRQ 号 ”介绍过的技术之一来自动检测。

“ 即插即用 ”规范

有些 ISA 设备板子遵循特殊的设计准则，要求特别的初始化序列，以简化增加接口板的安装和配置。这类板子的设计规范被称做“ 即插即用（PnP）”，它由一组构造和配置无跳线 ISA 设备的繁杂的规则集组成。PnP 设备实现了可重定位的 I/O 区段；PC 的 BIOS 负责这个重定位—--PCI 的风格。
简单地说，PnP 的目的就是 PCI 设备具有的同样的灵活性，而不改变底层的电气接口（ISA 总线）。为了这个目的，规范定义了一组设备无关的配置寄存器和地理寻址接口板的方法，即使物理总线并不携带每个板子（地理的）的走线----每个 ISA 信号线于每个可用槽相连。地理寻址工作的方式是 给计算机的每个 ISA 外围分配一个小整数，称做“ 卡选择号（CSN）”。每个 PnP 设备有一个唯一的序列标志符，64 位宽，被硬写入外围板子。CSN 的分配用这个唯一的序列号来确定 PnP 设备。但 CSN 只能在引导时被安全地分配，这要求 BIOS 理解 PnP。由于这个原因，如果没有一个配置盘，老计算机不能支持 PnP。
符合 PnP 规范的接口板在硬件级十分复杂。它们比 PCI 板要精细的多，同时要求符杂的软件。安装这类设备遇到困难并不罕见；即使安装没有问题，你仍然面对性能限制和 ISA 总线有限的 I/O 空间。按我的观点，只要可能，最好是安装 PCI 设备并享受其新技术。
如果你对 PnP 的配置软件有兴趣，你可以浏览 *drivers/net/3c509.c*，它的探测函数处理了 PnP 设备。Linux2.1.33 也为 PnP 增加了一些初始支持，见目录 *drivers/pnp*。

其它 PC 总线

PCI 和 ISA 是 PC 世界最常用的外围接口，但它们并不为仅有。下面是 PC 市场上找得到的其它总线特征的概述。

MCA

“ 微通道体系结构（MCA）”是用在 PS/2 计算机和一些笔记本上的一个 IBM 标准。微通道的主要问题是缺乏文档，这导致了 Linux 上对 MCA 支持的缺乏。不过，在 2.1.15，已经飘荡多时的 MCA 补丁被加入了正式的核心；因此，新的核心可以在 PS/2 计算机上运行。
在硬件级，微通道具有比 ISA 多的特征。它支持多主 DMA，32 位地址和数据线，共享中断

* 中断共享的问题是属于电气工程的；如果一个设备驱动程序驱动信号线为不活动----通过应用一个低阻抗电压级----中断便不能共享。另一方面，如果设备对不活动逻辑级使用一个上拉电阻，那么共享就是可能的。多数ISA接口板使用上拉的方法。

线，以及访问每个板子配置寄存器的地理寻址。这种寄存器被称做“ 可编程选项选择（POS）”，但它们并不具有 PCI 寄存器的所有特征。Linux 对 MCA 的支持包括一些引出到模块的函数。设备驱动程序可以通过读取整数值 MCA_bus 来确定它是否运行在一个微通道计算机上。如果核心运行在一个 MCA 单元中，那么 MCA_bus 非 0。如果这个符号是个预处理器宏，那么宏 MCA_bus__is_a_macro 也要被定义。如果 MCA_bus__is_a_macro 未定义，那么 MCA_bus 是引出到模块化代码的一个整数变量。事实上，MCA_bus 对除 PC 外的所有平台仍然是个硬写为 0 的宏----Linux X86 的移植在 2.1.15 将宏改为变量。MCA_bus 和 MCA_bus__is_a_macro 都在<asm/processor.h>中定义。

EISA

扩展的 ISA（EISA）是 ISA 的 32 位扩展，带一个兼容的接口连接器；ISA 的设备板子可以插入一个 EISA 连接器。额外的线路是在 ISA 连接下路由。

象 PCI 和 MCA，EISA 总线被设计来带无跳线设备，它与 MCA 有同样的特征：32 位地址和数据线，多主 DMA，以及共享中断线。EISA 设备由软件配置，但它们不需要任何特别的操作系统支持。EISA 驱动程序已经为以太网设备和 SCSI 控制器存在与 Linux 核心中。

EISA 驱动程序检查 EISA_bus 确定是否主机带有 EISA 总线。类似于 MCA_bus，EISA_bus 要么是宏，要么是变量，这依赖于 EISA_bus__is_a_macro 是否被定义了。这两个符号定义在<asm/processor.h>。

至于驱动程序，核心中没有对 EISA 的特别支持，程序员必须自己处理 ISA 的扩展。驱动程序用标准的 EISA I/O 操作访问 EISA 寄存器。核心中已有的驱动程序可以作为示例代码。

VLB

ISA 的另一个扩展是“ VESA Local Bus ”接口总线，它通过增加一个长度方向的槽扩展 ISA 连接器。这个额外的槽可以被 VLB 设备“ 单独 ”使用；由于它从 ISA 连接器复制了所有重要的信号，设备可以被构造成只插入 VLB 插槽，而不用 ISA 插槽。单独的 VLB 外围很少见，因为多数设备需要到达背板，这样它们的外部连接器是可用的。

Sbus

虽然多数 Alpha 计算机装备有 PCI 或 ISA 接口总线，多数基于 Sparc 的工作站使用 Sbus 连接它们的外围。

Sbus 是相当先进的一种设计，尽管它已经存在了很长时间了。它是想成为处理器无关的，并专为 I/O 外围板子进行了优化。换句话说，你不能在 Sbus 的槽中插入额外的 RAM。这个优化的目的是简化硬件设备和系统软件的设计，这是以母板上的额外复杂性为代价的。

总线的这种 I/O 偏向导致了一类外围，它们用虚地址传送数据，这样绕过了分配连续缓冲区的需要。母板负责解码虚地址，并映射到物理地址上。这要求在 Sbus 上附带一些 MMU（内存管理单元）的能力，这部分负责的电路被称为“ IOMMU ”。这种总线的另一个特征是：设备板子是地理寻址的，因此不需要在每个外围上实现地址解码器，也不需要处理地址冲突。Sbus 外围在 PROM 中使用 Forth 语言来初始化它们。选择 Forth 是因为这个解释器是轻量级的，可以容易地在任何计算机系统的固件里实现。另外，Sbus 规范描述了引导过程，因此符合条件的 I/O 设备可以简单地接入系统，并在系统引导时被识别出来。

至于 Linux，直到核心 2.0 也没有对 Sbus 设备的特别支持被引出到模块中。版本 2.1.8 增加了对 Sbus 的特定支持，我鼓励感兴趣的读者去看看最近的核心。

快速参考

本节象往常一样，概述在本章中介绍的符号。

#include <linux/config.h>

CONFIG_PCI

这个宏用来条件编译 PCI 相关的代码。当一个 PCI 模块被加载入一个非 PCI 核心时，*insmod* 会抱怨说几个符号不能解析。

#include <linux/pci.h>

这个头文件包含 PCI 寄存器和几个销售商及设备 ID 的符号名。

#include <linux/bios32.h>

所有下面列出的 pcibios_函数在这个头文件中定义原型。

```
int pcibios_present(void);
```

这个函数返回一个布尔值表明我们运行的计算机是否具有 PCI 能力。

```
int pcibios_find_device(unsigned short vendor, unsigned short id, unsigned short index,
                        unsigned char *bus, unsigned char *function);
int pcibios_find_class(unsigned int class_code, unsigned short index,
                       unsigned char *bus, unsigned char *function);
```

这些函数询问 PCI 固件关于设备是否有某个特定的签名，或属于某个特定的类。返回值是一个出错说明；成功时，bus 和 function 用来存储设备的位置。index 第一次必须被传给 0，以后没查找一个新设备，就增加 1。

PCIBIOS_SUCCESSFUL

PCIBIOS_DEVICE_NOT_FOUND

```
char *pcibios_strerror(int error);
```

这些宏，还有其它几个表示 pcibios 函数返回的整数值。DEVICE_NOT_FOUND 一般被认为是个成功值，因为成功地发现没有设备。pcibios_strerror 函数用来将每个整数返回值转换为一个字符串。

```
int pcibios_read_config_byte(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_word(unsigned char bus, unsigned char function,
                             unsigned char where, unsigned char *ptr);
int pcibios_read_config_dword(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned char *ptr);
int pcibios_write_config_byte(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned char val);
int pcibios_write_config_word(unsigned char bus, unsigned char function,
                              unsigned char where, unsigned short val);
int pcibios_write_config_dword(unsigned char bus, unsigned char function,
                               unsigned char where, unsigned int val);
```

这些函数用来读写 PCI 配置寄存器。尽管 Linux 核心负责字节序，程序员在从单个字节组装多字节值时必须特别注意字节序。PCI 总线是小印地安字节序。

第十六章　核心源码的物理布局

到目前为止，我们从写设备驱动程序的角度讨论了 Linux 核心。但一旦你开始研究核心，你会发现你想“全面理解它”。事实上，你可能整天在浏览源码，搜索源码树，目的只是想搞清楚核心不同部分之间的关系。

这种“沉重的搜索”是我在家里专门设一台计算机的任务之一，并且这是从源码中获取信息的一个有效的办法。然而，在坐在你喜欢的 shell 提示符之前若能得到一些知识基础将会很有帮助。本章基于版本 2.0.x，对 Linux 核心源文件提供一个快速的概览。文件布局在版本之间的改变并不大，尽管我不能保证将来会不会变。因此下面的信息对浏览核心的其它版本应该也很有用，即使它不是权威。

在本章中，每个给出的路径名都是相对于源码的根（通常是*/usr/src/linux*），而没有目录部分的文件名一般假设它居于“当前”目录----即正在讨论的哪个。头文件（当以角括弧的形式命名时----<和>）是相对于源码树的 *include* 目录给出的。我不想介绍 *Documantation* 目录，因为它的作用应该很清楚。

引导核心

看一个程序的一般方法是从执行开始的地方着手。至于 Linux，很难说执行是从那里开始的----它取决于你是如何定义“开始”的。

体系结构相关的开始点是 *start_kernel*，在 *init/main.c* 中。这个函数是从体系结构特定的代码中被调用的，但它并不返回到那里。它掌管着转动轮子，因此可以被认为是“所有函数的母亲”，计算机生命的第一次呼吸。在 *start_kernel* 之前是一片混沌。

在 *start_kernel* 被调用时，处理器已经被初始化了，保护模式（如果有）也被激活了，处理器在最高的优先级执行（通常被称为“管理员模式”），中断被关闭了。*start_kernel* 函数负责初始化所有的核心数据结构。这个通过调用外部函数来执行子任务，因为每个设置函数都在合适的核心子系统定义。*start_kernel* 也调用 *parse_options*(也在 *init/main.c* 文件中)来对从用户或引导系统的程序处传来的命令行进行解码。

命令行（与 *memory_start*，*memory_end* 一道）用 *setup_arch* 从计算机内存中获取。*setup_arch*，如它的名字提示，是体系结构特定的代码。

init/main.c 中的代码主要由#ifdefs 组成。这是因为初始化是按步发生的，很多步可能被运行或跳过，这依赖于核心的编译时配置。命令行的解释也严重地依赖于条件，因为很多参数只有在被编译的核心含有特定的驱动程序时才有意义。

由 *start_kernel* 调用的初始化函数有两种风格。一些函数没有参数，返回 void；而另一些需要两个 unsigned long 参数，并返回另一个 unsigned long 值。其参数是 memory_start 和 memory_end 的当前值，即未分配的物理内存的边界。返回值是新的 memory_start 值（如你所已知的，核心用 unsigned long 表达内存地址）。这个技术允许子系统在物理内存的开始处分配一个持续的（和连续的）内存区域，如在第七章“把握内存”中“playing dirty”一节中提到的。这种技术的最大的缺点是它只能在引导时使用，对那些需要用于 DMA 的巨大内存区段的模块并不可用。

初始化完成后，*start_kernel* 打印出旗帜字符串，包括 Linux 版本号和编译时间，接着通过调用 *kernel_thread* 派生出(fork)一个 init 进程。

start_kernel 函数接着以任务 0（所谓的“空闲”任务）的形式继续，并调用 *cpu_idle*，它是一个调用 idle 的无限循环。在这一点，SMP(对称多处理器)的工作方式略有不同，但我不打

算讲述这个不同。idle 函数的真正行为是体系结构相关的，对源码的简单搜索可以把你带到可以研究其功能的位置。

引导之前

在前一节，我把 *start_kernel* 看作第一个核心函数。不过，你可能对这点之前发生的事情感兴趣。

在 *start_kernel* 之前运行的代码是低级的，包含汇编码，因此你可能对其细节不感兴趣。不过，我将介绍一下固件（在 PC 世界称为 BIOS）将控制交给 Linux 后计算机中发生了什么。如果你对钻研低级代码没有兴趣，你可以直接跳到“ Init 进程 ”。下面提供了关于 Intel，Alpha，Sparc 引导代码的一些提示，因为这是我能访问的仅有的系统。（如果有人肯捐一些硬件，我将在下一版覆盖更多的平台）。

设置 X86 处理器

个人计算机是基于一个老的设计，后向兼容性一直有很高的优先级。因此，PC 固件还是以一种老方式引导操作系统。一旦引导设备被选择，它的第一个扇区被加载到内存的 0x7C00 处，然后让出控制。

刚加电的处理器处于实模式（也就是说，它象 8086）并只能寻址物理内存的前 640KB。其中一部分已经被固件管理的数据表格占用了。由于核心要比这个大，Linux 的开发者必须找到一个不一般的方法将核心影响加载到内存。结果就是 *zImage*，即核心的压缩映象，它可以被装入底端内存（但愿如此），并在进入保护模式后自解压缩到高端内存。

这样引导扇区发现它面对着五百字节的代码，和半兆字节的空闲内存。引导代码真正做的依赖于系统是如何引导的。引导扇区可以是第一个核心扇区（如果你直接从软盘上引导 *zImage*）或者 *lilo*。如果 Linux 由 *loadlin* 引导，则没有引导扇区什么事，因为在 *loadlin* 运行时，系统已经被引导了。

引导一个 bare-bones *zImage* 核心

如果被引导的系统是软盘上的核心映象，在引导扇区执行的代码是 *arch/i386/boot/boot.S*（一个实模式的汇编文件）。它将自己移到地址 0x90000，从可引导设备上加载另外几个扇区，把它们放在紧挨自己的后面（也就是 0x90200）。接着核心映象的其余部分被加载到地址 0x10000（64KB：固件数据空间之后）。

位于 0x90200 的代码是所谓的“ 设置 ”代码（*arch/i386/boot/setup.S* 和 *arch/i386/boot/video.S*），它负责各种硬件的初始化，以及对视频板子的初步检测，以便可以切换到不同的文本模式分辨率。这些任务在实模式中进行（使用 *loadlin* 时则是在 VM86 模式），因此可以使用 BIOS 调用，避免处理硬件特定的细节。

setup.S 接着把整个核心从 0x10000(64KB)移到 0x1000(4KB)；这样在核心代码之前只有一页被浪费了----这页其实并没有真的浪费；它在系统中自有它的用处。代码的这种来回复制是为了摆脱被 BIOS 强加的内存布局，还能不至于覆盖重要的数据。最后 *setup.S* 进入保护模式，跳转到 0x1000。

arch/i386/boot/compressed/head.S(用 *gas* 写成，因为我们已经在保护模式了)设置栈。接着调用 *decompress_kernel*，它把已解开的代码放在地址 0x100000(1M)并跳转到那里。

arch/i386/kernel/head.S 是被解压缩核心的头；它建立最后的处理器设置（与硬件换页有关的寄存器处理）并调用 *start_kernel*。这就是所有需要的----已经完成了。

引导一个 bare-bones *bzImage* 核心

随着越来越多的驱动程序为 Linux 核心开发出来，一个全特征的压缩核心不再能放入低端内存。例如，这种情况对安装核心就可能发生，因为它为了能适应各种配置，塞满了不同的驱动程序。因此，必须设计另一种加载方法。*bzImage* 就是大的 *zImage*，它在不能放入低端内存时也可以被加载。

有几种加载 *bzImage* 的办法，这取决于使用的引导加载程序（boot loader）。核心负责每种情况，现在我打算从原始软盘上是如何引导的。

一个 *bzImage* 核心的引导扇区不能简单地将所有的压缩数据加载到低端内存，所以它必须欺骗（如多数实模式 x86 程序所做的那样）。如果被加载的映象是大的，引导扇区象往常一样加载“ 设置 ”扇区，但在主引导循环的每次叠代都有一个“ 助手 ”例程被调用。助手例程在 *setup.S* 中定义，因为引导扇区太小无法放下它。这个例程用一个 BIOS 调用将数据从低端移到高端内存，一次移动 64KB，它还要重置目的地址，引导扇区用来从盘上传送下一次的数据。这样，在 *bootsect.S* 中的一般加载例程就不会用尽低端内存。

在核心被加载后，*setup.S* 象往常一样被调用。它除了改变上一个跳转指令的目的地址外，并不做任何特殊的工作。由于我们加载了一个大映象，处理器通过使用一台特殊的机器指令（它允许 386 在实模式段使用 32 位偏移）跳到 0x100000 而不是 0x1000。

解压缩和往常一样工作，但输出不能放在 0x100000（1M），因为压缩的映象已经在那儿了。解压的数据被写到低端的内存直到用尽；接着被写到越过压缩映象的地方。这两个解压的片段通过执行另外的内存移动在 0x100000 处装配起来。但复制例程也居于高端内存，它必须首先将自己复制到低端内存已防止被覆盖；然后它把整个映象移到 0x100000。

到这儿，游戏就结束了。但 kernel/head.S 并没有注意到发生的额外工作，所有事情照常进行。

使用 *lilo*

lilo，Linux 加载程序，居于引导扇区----或者是主引导扇区，或者是磁盘分区的第一个扇区。它使用 BIOS 调用从一个文件系统中加载核心。

这个程序与核心映象面对同样的问题：在机器引导时，仅有半 KB 的代码被装入内存，而且只用几打的指令解码一个文件系统结构也是不可能的。*lilo* 通过在安装时构造一个磁盘映射来解决这个问题。它用这个映射告诉 BIOS 从正确的地方获取每个核心块。这个技术很有效，但你在替换或者重写一个核心映象后必须重新安装 *lilo*----你必须调用 *lilo* 命令，用一个新的核心块表来重新安装引导加载程序。

实际上，*lilo* 扩展了加载机制，它允许用户在引导时选择加载哪个映象。这个选择是通过一个映象的安装定义表来做到的。它用从不同的分区中取出的引导扇区代替它自己的引导扇区来实现。

lilo 比一个 barebone 引导的最大好处（除了能从硬驱直接引导外）是它允许用户象核心传递一个命令行。这个命令行可以在 *lilo* 配置文件中指定，也可以在引导时交互给出。*lilo* 把命令行放在零页（我们将其在 *boot/head.S* 之前保持空闲）的后一半。这一页以后由 *setup_arch*(在 *arch/i386/kernel/setup.c* 中定义)取得。

lilo 的最近版本（18 版本甚至更新）可以加载 *bzImage*，而老的发布是不能的。较新的版本可以用 BIOS 调用将数据加载到高端内存，象 *bootsect.S* 做的那样。

当 *lilo* 完成加载，它跳到 *setup.S*，事情就象我们以前看到的那样继续进行。

使用 *loadlin*

*loadlin*用来将Linux从一个实模式操作系统中引导起来。与*lilo*类似，都是加载数据，传递命令行，跳至*setup.S*。但它有一个优点就是它可以在FAT分区中从一个指定的文件名加载核心，

而不需要一个块的映射。这使得它比较稳定。如果你想加载*bzImage*，你需要*loadlin*的版本1.6或更新*。有趣的是注意到*loadlin*可能需要玩一些脏活才能加载整个核心，同时又不至于搞乱宿主操作系统。只有在核心的所有部分都被加载了，*loadlin*才能在合适的地址重新装配它，并调用它的入口点。

其它引导方式

还有一些程序可以引导 Linux 核心。其中的两个是 *Etherboot* 和 *syslinux*，当然还有很多。不过我不打算在这里讲述它们，因为它们与我已经讲过的类似，至少与核心相关的部分如此。但要注意，引导一个 Linux 核心并不是象我说的这么简单。要进行大量的检测，版本号经常出现在特别的地方，以抓住用户的错误，并友好的回复。意思是如果发生了什么问题，系统可以在挂起前打印一条信息。局限在 x86 实模式的执行环境下很难完全避免发生错误时挂起，打印一条消息总比什么都没有强。

设置 Alpha 处理器

让一个 Alpha 到达能运行 *start_kernel* 这一点要比 Intel 处理器容易的多，因为在 Alpha 上不需要和实模式或内存限制做斗争。而且，Alpha 工作站通常配有比 PC 好的固件，可以从文件系统装载一个完整的文件。我不想讨论装载一个文件时的实际步骤，因为这个代码没有随 Linux 发布，这样你无法检查它----我也不能，因此就无法谈论它。

milo（迷你加载程序）程序是引导的一般选择。*milo* 比固件要聪明，因为它理解 Linux 和它的文件系统，但又比核心笨，因为它不能运行进程。*milo* 由固件从 FAT 分区执行，可以从 ext2 或 ISO9660 块设备上加载核心。象 *lilo* 和 *loadlin*，*milo* 也向核心传递一个命令行。在 Linux 被加载到内存中正确的虚地址后，*milo* 转向核心，自己消失。

milo 的有些特征依赖于核心源码，因为它需要访问设备，理解文件系统布局。配有驱动程序和文件系统类型，它可以根据文件名从硬盘或 CD-ROM 上取得核心映象。这个设计后面的想法与 *loadlin* 类似，只是 *milo* 使用 Linux 核心的代码，而不是取自别的操作系统环境。

在 Alpha 上引导 Linux 并不总是可用 *milo*。如果你的系统有 SRM 固件，就不能安装 *milo*。相反，你可以使用 arch/alpha/boot 中的原始加载程序。这个加载程序很简单，能从硬盘或软驱中读取一个顺序区域，这与 PC 上 *zImage* 前面的引导扇区所做的工作一样。使用原始加载程序要求核心映象必须（在任何文件系统之外）被复制到磁盘上的连续区域。

如果不考虑系统是如何引导的，控制被传递给 *arch/alpha/kernel/head.S*，但 Linus 说：“ 没什么需要我们做的了 ”。源码只是设置几个指针，然后就跳到 *start_kernel*。

设置 Sparc 处理器

Sparc 计算机用一个称为 *silo* 的程序引导 Linux。与 *lilo*，*milo* 命名方法类似，只是用“ s ”表示 Sparc。引导 Sparc 比 Alpha 要简单一些；它的固件可以访问设备，*silo* 只需要访问 Linux 文件系统，并与用户交互。出于这个目的，*silo* 被链接到 *libext2*，这是支持对未安装分区上的文件进行处理的一个库。

若不使用 *silo*，也可以从软盘或网络上引导计算机。固件可以用 RARP（反向 ARP）和 tftp 协议从以太网上装载一个核心。事实上，我从未用软盘引导过我自己的工作站，因为 Linux 的 Sparc 发布允许通过网络引导来完成系统安装。

对 Sparc 来说，的确没有什么特别的要求。没有实模式，也没有需要复制的内存。一旦核心被加载到 RAM，它便开始执行。

---

* 你可能需要 1.6a或更新来加载 2.1.22 或更新的核心。

Init 进程

由 *start_kernel* 生成的线程派生出 bdflush（源码见 *fs/buffer.c*）和 kswapd（在 *mm/vmscan.c* 中定义），它们因此被赋予进程号 2 和 3。*init* 进程（pid 1）接着进行进一步的初始化，这在之前不可能完成；也就是，它运行与 SMP 相关的函数，如果需要，还有 initrd 引导技巧，以另一个核心线程的形式。在 initrd 结束后，init 线程激活 UMSDOS 文件系统的“伪根(pseudo-root)”。

在完成初始化后，*init* 的实际作用是进入用户空间并执行一个程序（因此变成一个进程）。这样三个 stdio 通道被连到第一个虚拟控制台，核心试图从/etc 执行 *init*。如果失败，它将查看*/bin* 和*/sbin*（在所有最近的发布中，*init* 一定居于此）。如果 *init* 从这三个目录的执行都失败了，进程将会执行*/etc/rc*，如果这个也失败了，它就循环，执行*/bin/sh*。在大多数情况下，函数能运行 *init* 成功；其它选项的目的是为了在 *init* 不能执行时允许系统恢复。

如果核心命令行指定了一条要执行的命令，使用 init=*some_program* 导语，进程 1 就执行指定的命令，而不是调用 *init*。

不管系统是怎么设置的，*init* 最终在用户空间执行，以后的核心操作都是对来自用户程序的系统调用的响应。

kernel 目录

大多数关键的核心功能都是在这个目录实现的。这里最重要的源文件是 *sched.c*，它值得特别对待。

sched.c

正如源文件自己表明的，这是“主核心文件”。它由调度程序和相关操作组成，例如让进程睡眠和唤醒它们，以及核心计时器的管理（见第六章“时间流”中“核心计时器”一节），间隔计时器（它与计帐和性能刻划有关），以及预定义任务队列（见第六章“预定义任务队列”）。

如果你对 Linux 调度程序的实时策略感兴趣，你可以在 schedule 函数及其相关者中找到低级的信息。其中一个相关者是 goodness，它给进程赋优先值，并帮助调度程序选择下一个要运行的进程。

与调度程序控制相关的函数（及系统调用）也在这个文件中定义。这包括设置和取得调度策略及优先级。在除 Alpha 外的其它体系结构上，系统调用 *nice* 也在这个源文件中。

另外，取得和设置用户及组id也在*sched.c*中定义（除了Alpha），同时还有*alarm*调用*。

在 *sched.c* 中还能找到的其它好东西包括 *show_tasks* 和 *show_state* 函数，它们实现了在第四章“调试技巧”中“系统挂起”一节所描述的“魔幻”键中的两个。

进程控制

目录的其它主要部分都是管理进程的。*fork* 和 *exit* 系统调用在两个同名源文件中实现，信号控制在 *signal.c* 中实现。大多数信号处理的调用在 Alpha 中实现的方法是不同的，以保证 Alpha 的移植与 Digital Unix 二进制兼容。

fork 的实现包括 clone 系统调用的代码，*fork.c* 显示了 *clone* 的标志是如何使用的。应该注意 *sys_fork* 并不在 *fork.c* 中定义，因为 Sparc 的实现与其它的版本稍有不同；不过，多数 *sys_fork* 的实现只是调用 *do_fork*，它在 *fork.c* 中定义。提供一个缺省实现（通常叫做 *do_fnct*），而真

正的系统调用（*sys_fnct*）则在各个移植中声明，这是 Linux 常用的一个技巧，随着新的移植的出现，这个技巧很可能扩展到其它的系统调用。
exit.c 实现 *sys_exit* 和不同的 *wait* 函数，以及信号的实际发送。（*signal.c* 专用于信号处理，而不是发送。）

模块化

文件 *module.c* 和 *ksyms.c* 包含了在第二章“ 构造和运行模块 ”中描述的机制。*module.c* 含有被 *insmod* 及相关程序使用的系统调用，*ksyms.c* 声明不属于特定子系统的核心中的公共符号。其它的公共符号由特定核心子系统的初始化函数使用 *register_symtab* 声明。例如，*fs/proc/procfs_syms.c* 为注册新文件声明*/proc* 接口。

其它操作

这个目录中的其余源文件为一些低级操作提供软件接口。*time.c* 从用户程序读写核心时间值，*resource.c* 为 I/O 端口实现请求和释放机制，*dma.c* 为 DMA 通道完成同样的工作。*softirq.c* 处理下半部（见第九章“ 中断处理 ”中“ 下半部 ”一节），*itimer.c* 定义系统调用来设置和取得间隔计时器值。
想知道核心的消息处理是如何工作的，你可以看 *printk.c*，它显示了在第四章介绍的几个概念的一些细节（也就是说，它包含了 *printk* 和 *sys_syslog* 的代码）。
exec_domain.c 包含了获得与其它风格的 Unix 兼容性所需要的代码，*info.c* 定义了 *sys_info*。*panic.c* 做的工作正如它的名字所示；它还支持在系统不稳定后自动重启动。重新引导发生在由*/proc/sys/kernel/panic* 设置的延迟之后。这个延迟通过对 udelay(1000)的重复调用实现，因为在系统崩溃后，调度程序不再运行，udelay 可以用于不长于 1 毫秒的延迟（见第六章“ 长延迟 ”一节）。
sys.c 实现几个系统配置和权限处理函数，如 *uname*，*setsid* 及类似的调用。*sysctl.c* 包含 *sysctl* 调用的实现和在 sysctl 表（系统控制入口点列表）注册及取消注册的入口点。这个文件也提供了按照注册的表访问*/proc/sys* 文件的能力。

mm 目录

在 mm 目录中的文件为 Linux 核心实现内存管理中体系结构无关的部分。这个目录包含换页及内存的分配和释放的函数，还有允许用户进程将内存区间映射到它们地址空间的各种技术。

换页和对换

令人惊奇的是，*swap.c* 并未实现对换算法。相反，它处理核心的命令行选项 swap=和 buff=。这些选项也可以通过 *sysctl* 系统调用或写文件*/proc/sys/vm* 来设置。
sawp_state.c 负责维护对换高速缓存，是这个目录中最难的文件；我不想讨论它的细节，因为很难理解它的设计，除非以前对相关的数据结构和策略已经有了很好的了解。
swapfile.c 实现对换文件和设备的管理。*swapon* 和 *swapoff* 系统调用在这里定义，后者代码非常困难。作为比较，有几个 Unix 系统没有实现 *swapoff*，这样如果不重启动就无法停止向一个设备或文件的对换。*swapfile.c* 还声明了 *get_swap_page*，它从对换池中取得一个空闲页。
vmscan.c 实现换页策略。*kswapd* 守护进程在这个文件定义，还有扫描内存，运行进程寻找可换出的页的函数。

---

* 这个调用在当前的libc版本中不再使用，它通过计时器的方式实现这个函数。

最后，*page_io.c* 实现了与对换空间之间进行低级数据传送的功能。这个文件管理保证系统一致性的锁机制，提供同步和异步的 I/O。它还处理与不同设备使用不同块大小相关的问题。（在 Linux 的早期版本，不可能对换到一个 FAT 分区上，因为不支持 512 字节的块。）

分配和释放

再第七章介绍的内存分配技巧都在 *mm* 目录实现。让我们再一次从最常用的函数开始：*kmalloc*。

kmalloc.c 实现内存区域的分配和释放。*kmalloc* 的内存池由一些“桶”组成，每个桶是同样大小的内存区域的列表。*kmalloc.c* 的主要功能是管理每个桶的链表。

当需要新页或有页面被释放时，这个文件利用在 *page_alloc.c* 中定义的函数。页面用 *__get_free_pages* 从空闲内存中取得，这是一个从空闲页列表中取得页面的短函数。如果空闲列表中没有内存可用，就调用 *try_to_free_pages*(*vmscan.c*)。

vmalloc.c 实现了 *vmalloc*，*vremap*，*vfree* 函数。*vmalloc* 返回核心虚拟空间中的连续内存，*vremap* 给出特定物理地址的新的虚地址；它主要用来访问高端内存的 PCI 缓冲。*vfree* 释放内存，如它的名字所示。

其它接口

Linux 内存管理最重要的函数是 *memory.c* 文件的一部分。这些函数一般不能通过系统调用访问，因为它们处理硬件的换页机制。

另一方面，模块的作者的确使用这些函数。*verify_area* 和 *remap_page_range* 在 *memory.c* 中定义。其它有趣的函数是 *do_up_page* 和 *do_no_page*，它们实现核心对次和主页面错的反应。文件中的其余函数处理页表，都非常低级。

内存映射是 *mm* 目录中文件处理的另一个大任务。*filemap.c* 的代码很复杂。它实现常规文件的内存映射，提供支持共享映射的能力。被映射文件通过被映射页的特殊结构 vm_operations 支持，如在十三章“Mmap 和 DMA”中“虚拟内存区域”一节中所描述的。这个源文件页处理异步提前读；注释解释了结构 file 中四个提前读域的含义。这个文件中出现的唯一的系统调用是 *sys_msym*。到内存映射的顶级接口（即 *do_mmap*）出现在 *mmap.c*。

这个文件有定义 *brk* 系统调用开始，它被一个进程用来请求其最高允许的虚地址被增加或减小。*sys_brk* 的代码提供了很多信息，即使你不是内存管理的大师。*mmap.c* 的其余部分集中在 *do_mmap* 和 *do_munmap*。如你所期望的，内存映射通过 filp->f_op 完成，尽管 filp 对 *do_mmap* 可能为 NULL。这是 *brk* 如何分配新的虚拟空间的。它还是依赖于内存映射零页，而不需要特殊代码。

mremap.c 包括 *sys_mremap*。如果你已经搞清楚了 *mmap.c*，这个文件就很容易了。

与内存锁定和解锁相关的四个系统调用在 *mlock.c* 中定义，它是相当简单发源文件。类似第，*mprotect.c* 负责执行 *sys_mprotect*。这些文件在定义上很相似，因为它们都修改了与进程页相关的系统标志。

fs 目录

在我的观点中，这个目录是整个源码树中最有趣的一部分。文件处理是任何 Unix 系统的一个基本活动，所有与文件相关的操作都在这个目录中实现。我不想在这儿描述 *fs* 子目录，因为每种文件系统类型仅仅是把 VFS 层映射到特别的信息布局上。但讲一下 *fs* 目录中多数文件的作用还是很重要的，因为它们携带了大量的信息。

Exec 和二进制格式

Unix 中最重要的系统调用是 *exec*。用户程序可以 *exec* 六种不同的形式，但在只有一个在核心中实现----其它都是隐射到全特征实现 *execve* 上的库函数。

exec 函数用一个已注册二进制格式的表以查找用来加载和执行一个磁盘文件的正确的加载程序。源文件的第一部分定义了 *register_binfmt* 接口。有兴趣的是注意到脚本文件的#!魔幻键被作为二进制格式处理，如 ELF 和另外一些格式（尽管在 1.2 版本中它是 *exec.c* 的一种特殊情况）。*kerneld* 也需要了解一个新的二进制格式是如何按照要求从读取源文件被加载的。

每个二进制格式由一个定义了三种操作（加载一个二进制文件，加载一个库，倾倒内核（core dump））的数据结构描述。这个结构在<linux/binfmts.h>中定义。决大多数格式都只支持第一个操作（加载一个文件），不过这个接口已经足够一般化，能够支持任何可预见的新格式的需要。

devices.ch 和 block_dev.c

我们已经用了 *devices.c* 中的代码，因为它负责设备注册和取消注册。它同时也负责缺省的设备 *open* 方法，以及对块设备的 *release* 方法。这些调用为被打开或关闭的设备取得正确的文件操作并将执行分派到正确的方法。对模块自动加载的支持在这个文件中实现。除了打开和关闭设备之外的所有东西都出现在 *drivers/**目录，如 filp->f_op 所指示的。

block_dev.c 包含读写块设备的缺省方法。如你可能记得的，一个块驱动程序并不声明它自己的 I/O 方法，只是它的请求例程。*block_dev.c* 的缺省读写实现了解缓冲高速缓存，除了实际的数据传送它提你做了所有的事情。

VFS:超级块

程序和设备的执行只是 *fs* 目录的一部分。*fs* 的多数文件，以及子目录中的所有文件，都与文件相关的系统调用有关。更特别地：它们实现了所谓的 VFS 机制：虚拟文件系统（或虚拟文件系统切换----这些解释有点互相矛盾）。

概念地说，VFS 是 Linux 文件处理软件的一层。通过利用各种文件系统格式提供的特征，这层提供了到文件的统一接口。在磁盘上布局信息的各种技巧可以通过 VFS 接口用一致的方法访问。实际上，VFS 减少到几个定义“ 操作 ”的结构。每个文件系统声明处理超级块、inode、和文件的这些操作。我们在本书中已经使用的 file_operations 结构就是 VFS 接口的一部分。

核心通过安装每个文件系统来访问它。*mount* 的一个任务是从磁盘上搜取所谓的“ 超级块 ”。超级块是一个文件系统中的主要数据结构。它的名字来自于一个事实，历史上，它曾经是磁盘上的第一个物理块。文件 *super.c* 包括与超级块有关的有趣操作的源码：读取的同步它们、安装和卸装文件系统、在引导时安装根文件系统。

除了这些有趣的（还有点复杂）操作，*super.c* 也返回与文件系统有关的信息，包括由 */proc/mounts* 和*/proc/filesystems* 提供的信息。

函数 *register_filesystem* 和 *unregister_filesystem* 被模块化的文件系统类型使用；它们也在 *super.c* 中定义。文件 *filesystems.c* 是一个#ifdef 语句的短表。依赖于那些选项被编译进核心，对应不同文件系统的各种 *init* 函数被调用。每个文件系统类型的 *init* 函数调用 *register_filesystem*，因此不需要别的条件编译选项。

Inode 和高速缓存技术

VFS 接口的下一块是 inode。每个 inode 由一个由设备号和 inode 号组成的唯一的键值确定。用户程序用文件名去访问文件系统中的结点，核心负责将文件名映射到唯一的键值。为了获

得更好的性能，Linux 维护了两个与 inode 键值相关的高速缓存：inode 高速缓存和名字高速缓存（也叫目录高速缓存）。另外，核心还负责已经熟悉的缓冲高速缓存。
inode 高速缓存是一个哈希表，用于从设备/ inode 号键值查找 inode 结构。高速缓存的实现，以及读写 inode 的例程，都在 *inode.c* 中。这个文件也实现了 inode 结构的锁机制以防止可能的死锁。
名字高速缓存是一个表，它将 inode 号和文件名关联起来。当一个名字被连着用了几次，高速缓存就可以避免重复的目录查询。源文件 *dcache.c* 包含管理高速缓存的软件机制。使用名字高速缓存的多数系统调用和函数是 *namei.c*（表示 name-inode）的一部分，包括 *sys_mkdir*，*sys_symlin*k，*sys_rename*，及类似的调用。
缓冲高速缓存是系统中最大的数据高速缓存，它的实现出现在巨大的文件 *buffer.c* 中。
至于文件，*file_table.c* 负责 file 结构的分配和释放。这包括 *get_empty_filp*，它被 *open*，*pipe*，*socket* 调用。

open.c

fs 中其它源文件中多数负责文件操作----与在驱动程序中需要实现的一样。第一个这样的文件是 *open.c*，它包括了很多系统调用的代码。它也包括 *sys_open*，及它的低级的对应者 *do_open*，还有 *sys_close*。这些系统调用都很直接，被映射到 filp->f_op。
open.c 包含修改 inode 的系统调用：*chown* 和 *chmod*，以及它们的对应者 *fchown* 和 *fchmod*。如果你对安全检查和不可变标志的使用感兴趣，你可以浏览源码，它可以被几乎所有的 Unix 编程者理解。改变 inode 中的次数也被支持----*utime* 和 *utimes* 在这里定义。
chroot，*chdir*，和 *fchdir* 也在 *open.c* 中找到，同时还有其它“ 改变 ”函数。
源码中定义的第一个函数是 *statfs* 和 *fstatfs*，它们通过 inode->I_sb->s_op->statfs 被分派文件系统特定的代码。
truncate 和 *access* 调用也出现在这个文件中。后者用进程的实际 uid 和 gid 检查文件的权限，暂时不考虑 fsuid 和 fsgid。

read.c 和 *readdir.c*

正如其名字所示，*read_write.c* 包含读和写，但它也包含了 *lseek* 和 *llseek*（有人猜测这个前导 l 的数目是每十年增加一个）。*lseek* 是使用 off_t(long)的标准调用，而 *llseek* 使用 loff_t(long long)。*llseek* 系统调用映射到 *lseek* 文件操作，它被作为 *lseek* 的超集实现。有趣的是注意到核心 2.1 版将这个方法改名为 *llseek*，以与其实现保持一致。
read 和 *write* 是很简单的函数，因为实际的数据传送是通过 filp->f_op 来分派的；*read_write.c* 也包含了 *readv* 和 *writev* 的代码，它们稍为复杂，因为多块数据的传输必须跨过核心和用户边界。
Linux 不允许你直接读一个目录文件，如果你尝试，将会返回-EISDIR。目录只能用 *readdir* 系统调用来读取，它是文件系统无关的；或使用 *getdents*，即“ 取得目录项 ”。用 *getdents* 读一个目录要快一点，因为一个调用可以返回很多目录项，而 *readdir* 一次只能返回一项。不过，*getdents* 只被 libc-5.2 或更新的库所支持。

select.c

select 的完整实现居于 *select.c*，除了 *select_wait* 线入函数。尽管 *select.c* 的代码很有趣，但却很难阅读，因为数据结构太复杂（这在第五章“ 增强的字符设备驱动程序操作 ”中“ 底层数据结构中讨论过 ”）。不过 *select.c* 倒是阅读核心代码的一个好的起始点，因为它是自包含的；除了一些无关紧要的细节外它不依赖与其它源文件。

Pipe 和 fifos

pipe 和 fifo 通信通道的实现与字符设备驱动程序非常类似。通过为两个通道使用同样的文件操作来避免了代码重复；只有 *open* 方法不一样。除了 *fifo_open* 和 *fifo_init* 外所有的函数都在 *pipe.c* 中定义。由于 fifo 与文件系统结点类似，除了 file_operations 结构，它们还需要一组 inode_operations 与之相关。正确的结构在 *fifo.c* 中定义。

下一个有趣的事情是注意到在 pipe 和 fifo 的实现中，*pipe.c* 定义了两个 file_operation 结构：一个是为通道的可读侧的，一个是为可写侧的。这允许在 *read* 和 *write* 中跳过权限检查。

控制函数

文件的“ 控制 ”系统调用在两个根据这个调用命名的文件中：*fcntl.c* 和 *ioctl.c*。前者基本上是自包含的，因为为 *fcntl* 定义的所有的命令都是预定义的。由于其中一些只是 *dup* 的包裹者，因此 *dup* 的实现也在 *fcntl.c* 中。另外，由于 *fcntl* 调用负责异步触发，*kill_fasync* 也可以在此找到。

ioctl.c 包含 *ioctl* 系统调用的外部接口。它是一个短文件，当它收到不认识的命令时，还要依赖于文件操作。

文件锁

Linux 中实现了两类锁接口，*flock* 系统调用和 *fcntl* 命令。后者是 POSIX 兼容的。

文件 *locks.c* 包含了处理两个调用的代码。它也包括对强制锁的支持，它在 2.0.0 之前是一个编译选项，在 2.1.4 被改为一个安装（mount）时选项。

次要文件

fs 的文件还支持磁盘定额（quota）。*dquot.c* 实现了定额机制，而 *noquot.c* 包含空函数；如果定额没有被包含在核心的配置中，它就代替 *dquot.c* 被编译。

最后，*stat.c* 实现了 *stat*、*lstat*、*readlink* 系统调用。在 2.0.x 定义了 *stat* 和 *lstat* 的两个实现，以保持与旧的 x86 库的后向兼容。

网络

Linux 文件体系中的 net 目录是套接字抽象和网络协议的容库；这些特征使用了大量的代码，因为 Linux 支持集中不同的网络协议。每个协议（IP，IPX 等等）都居于它自己的子目录中。Unix 域的套接字被以一种网络协议对待，它们的实现可以在 *unix* 子目录找到。有趣的是注意到 2.0 版的核心只包含了 IPv4，而版本 2.1 包含了相当完整的 IPv6 的支持，新来的标准解决了 IPv4 的编号问题。

Linux 中的网络实现是基于用于设备文件的同样的文件操作。这很自然，因为网络连接（套接字）是用一般文件描述符描述的。在核心中一个套接字是由一个结构 socket(<linux/net.h>) 描述。文件 *socket.c* 是套接字文件操作的容库。它通过结构 proto_ops 分派系统调用到其中的一个网络协议。这个结构被每个网络协议定义以将系统调用映射到低级的数据处理。

net 下的每个目录（除 *bridge*，*core*，*ethernet*）都专用来实现一个网络协议。目录 *bridge* 包含符合 IEEE 规范的以太网桥的优化实现。*core* 中文件实现了通用的网络特征如：设备处理，火墙，选播和异名；这包括独立于底层协议（*core/sock.c*）的套接字缓冲区(*core/skbuff.c*)和套接字操作的处理。最后，*ethernet* 包含通用的以太网函数。

几乎每个 *net* 下的目录都有一个处理系统控制的文件；这样引出的信息可以通过*/proc/sys* 文

件树或通过 *sysctl* 系统调用来访问。到 *sysctl* 的核心接口允许对系统控制入口点动态的增加和去除，在<linux/sysctl.h>中定义。

IPC 和 lib 函数

进程间通信和库函数各有一个小的专用目录。
ipc 目录包含一个名为 *util.c* 的通用文件，以及每种通信方式的一个源文件：*sem.c*，*shm.c*，*msg.c*。*msg.c* 负责消息队列和 *kerneld* 引擎，*kerneld_send*。如果 IPC 没有在编译时打开，*util.c* 引出一些空函数，它们通过返回-ENOSYS 来实现 IPC 相关的系统调用。
库函数就象你在 C 程序中常见的一些工具和变量：*sprintf*，*vsprintf*，errno 整数值，以及被各种<linux/ctype.h>宏使用的_ctype 数组。文件 *string.c* 包含字符串函数的可移植实现，但只有在体系结构特定的代码不包含优化的线入函数时才能编译。如果线入函数在头文件中定义，那么在 *string.c* 中的实现应该用#ifdef 语句排除在外。
lib 中最“有趣”的文件是 *inflate.c*，它是 *gzip* 的“gunzip”部分，是从 *gzip* 本身展开从而允许在引导时使用压缩的 RAMdisk。这个技巧在需要的数据除了压缩，不能在一张软盘上放下时使用。

Drivers

现在对 Linux 的 *drivers* 目录已经没什么可说的了。这个目录中的源文件在整部书里已经被广泛引用；这也使我为什么在讨论源文件树时把它们留在最后。

字符设备、块设备、和网络驱动程序

尽管这些目录中大多数驱动程序是特定于某种特别的硬件的，还是有几个文件在系统设置时起到比较通用的作用。
关于目录 *drivers/char*，实现 N_TTY 行规则的代码就在这儿。N_TTY 是系统 tty 的缺省行规则，它在 *n_tty.c* 中定义。在 drivers/char 中的另一个设备无关的文件是 *misc.c*，它提供对“杂类”设备的支持。一个“杂类”设备是有一个次设备号的简化的字符设备驱动程序。
这个目录还包含了对 PC 控制台的支持，和其它一些体系结构相关的驱动程序；它实际上包含了一些在其它地方都不合适的杂项的集合。
drivers/block 则要清晰多了。它包含了多数快设备驱动程序的单个文件驱动程序，和全特征的 IDE 驱动程序，它被劈成了多个文件。这个目录的资格文件提供了通用目的的支持；*genhd.c* 处理分区表，*ll_rw_block.c* 负责与物理设备之间数据传送的低级机制。request 结构是 *ll_rw_block.c* 的主要部分。
drivers/net 包含了 PC 网卡驱动程序的长列表，以及几个其它体系结构的驱动程序（例如，*sunlance.c* 是为了多数 Sparc 计算机的接口的）。有些驱动程序比它看起来要复杂一些，这在第十四章“网络驱动程序”中介绍过。例如 *ppp.c* 声明自己的行规则。
在 *drivers/net* 下的通用目的源文件是 *Space.c* 和 *net_init.c*。*Space.c* 主要是包含了一个可用网络设备的表。这个表包含了一个#ifdef 项的长列表，它们在系统引导时被检差以检测和初始化网络设备。*net_init.c* 包含 *ether_setup*，*tr_setup*，和类似的通用目的函数。

SCSI 驱动程序

如在第一章“Linux 核心介绍”中所提到的，Linux 中的 SCSI 驱动程序没有被包含在一般的字符和块设备类中。这是因为 SCSI 接口总线有它自己的标准。因此，将 SCSI 设备和其它

驱动程序分开，开发者可以分离和共享公用代码。
drivers/scsi 中多数文件是为特定 SCSI 控制器的低级驱动程序。通用目的的 SCSI 实现在文件 *scsi_*.c* 中定义，还有 *sd.c* 是磁盘支持，*sr.c* 是 CD-ROM 支持，*st.c* 是 SCSI 磁带支持，*sg.c* 是通用 SCSI 支持。最后一个源文件对使用 SCSI 协议的设备定义了通用目的支持。扫描仪和其它通用设备可以由用户程序通过使用*/dev/sg**设备结点来控制。

其它子目录

别的硬件驱动程序由它们自己的子目录。*drivers/cdrom* 包含即不是 IDE 也不是 SCSI 的光驱的驱动程序。它们是常规的块设备驱动程序，有它们自己的主设备号。
drivers/isdn 是（如其名字所示）Linux 的 ISDN 实现；*drivers/sound* 是最常用的 PC 声卡板子的驱动程序集合。*drivers/pci* 包含了一个文件，它包含了所有已知 PCI 设备（销售商/ID 对）的列表。实际的 PCI 功能不在此定义，而是在体系结构特定的目录。
最后，*sbus* 包含了一个 *char* 子目录，和 Sparc 体系结构的控制台代码。随着 2.1 开发的进行，这个目录增长的很快。

体系结构相关性

Linux 核心的 2.0 及更新版具有相当好的跨平台可移植性，也就是说大多数代码可以不加区别地在所有支持的体系结构上运行。到目前为止，我们所看到的所有东西都是与硬件平台完全无关的。
arch 目录树是 Linux 核心中包含平台特定代码的一小部分。每个系统相关的函数都在每个 *arch* 子目录中被复制，因此所有子目录的结构都是类似的。这些子目录中最重要的是 *kernel*，它存放了与主要 *kernel* 源码目录相关的每个系统特定的函数。
有两个汇编源文件总能在 *kernel* 下找到。*head.S* 是在系统引导时执行的启动代码；它有时包含一些例外处理代码。另一个文件是 *entry.S*，它包含了到核心空间的入口点。特别地，每个这样的文件包含其自己体系结构的 sys_call_table ；每个体系结构有一个不同的表将系统调用号和函数关联起来。
另一个常常找到的子目录是 *lib*，它包含网络包的优化的校验和例程，有时还包含别的低级操作，如串操作；还有 *mm*，它进行页面错的低级处理（*fault.c*），以及在系统引导时的初始化代码（*init.c*）；还有 *boot*，它包含启动系统的代码。如你可能想象的，*i386/boot* 是 *boot* 子目录中最复杂的部分。
我不打算描述体系结构特定的代码，因为阅读它们并不有趣，而且它们往往充斥着汇编语句。为了理解这些代码，你首先得知道目标体系结构的一些细节。我不论如何也不会觉得阅读体系结构特定的函数有什么乐趣，因为它们是系统中肮脏的部分，处理大量硬件小问题。核心的其它部分才是有趣的。

第十七章 最新进展

Linux 一直在迅速地发展着，开发人员总是迫切希望改善核心内部，它们并不考虑向后兼容性。这种自由开发导致了不同版本核心提供的设备驱动程序接口之间一定程度的不兼容。不过，在应用级还保持着兼容，除了个别需要与核心特征进行低级交互的应用（象 *ps*）。
另一方面，设备驱动程序是直接链接到核心映象上的，因此必须与数据结构、全局变量、以及由内核系统引出的函数发生的改变保持一致。在开发过程中，随着新特征的加入，内部被修改；新的实现取代了就的实现，因为实践证明它们更快，更清晰。尽管不兼容性要求程序员在写模块时要做一些额外的工作，我认为连续的开发是 Linux 社区的成功点：严格的先后兼容性最终证明是有害的。
这一章讲述 2.0.*x* 和 2.1.43 之间的不同，这些将会与即将推出的 2.2 发布类似。Linus 在前几个 2.1 版本中引入了最重要的改变，这样核心就可以多经历几个 2.1 版本，使得驱动程序的作者有足够的时间在开发被锁定以发布稳定的 2.2 之前来稳定驱动程序。下面的小节介绍驱动程序是如何处理 2.0 和 2.1.43 之间的不同的。我已经修改了本书介绍的所有示例代码，使得它们可以同时在 2.0 和 2.1.43 上编译和运行，以及这之间的大多数版本。
驱动程序的新版本可以从 O'Reilly 的 FTP 站点上在线例子的 v2.1 目录下得到。2.0 和 2.1 之间的兼容性通过头文件 *sysdep-2.1.h* 获得，它可以与你自己的模块集成。我选择不把兼容性扩展到 1.2 避免了给 C 代码加载太多的条件，而且 1.2-2.0 的不同已经在前面的章节解释过了。在我将要写完这本书时，我了解到从 2.1.43 起又引入了一些小的不兼容性；我不打算对之加以评述，因为我不能保证对这些最新版本的完全支持。
注意在本章我不会讲述 2.1 开发系列引入的所有新东西。我要做的只是移植 2.0 模块，使之可以在 2.0 和 2.1 核心上运行。利用 2.1 的特征意味着放弃对不具有这些特征的 2.0 发布的支持。2.0 版本仍是本书的重点。
在写 *sysdep-2.1.h* 时，我已努力使你熟悉新的 API，我引入的宏用来使 2.1 的代码可以在 2.0 上跑，而不是相反。
本章以重要性逐渐降低的顺序介绍不兼容性；最重要的不同首先被介绍，次要的细节则在后面介绍。

模块化

在 Linux 社区中，模块化变的越来越重要，开发人员决定用一个更清晰的实现取代旧的。头文件<linux/module.h>在 2.1.18 中完全重写了，一个新的 API 被引入。如你所期望的，新的实现比旧的要容易使用。
为了加载你的模块，你将需要包 *modutils-2.1.34* 甚至更新版本（细节见 *Documentation/Changes*）。当与旧的核心一起使用时，这个包可以回到兼容模式，因此你可以用这个新包替换 *modules-2.0.0*，即使你经常在 2.0 和 2.1 之间切换。

引出符号

符号表的新接口比以前的要容易多了，它依赖于下面的宏：
EXPORT_NO_SYMBOLS；

这个宏与 register_symtab(NULL)等价；它可以出现在一个函数的内部或外部，因为它只是指导汇编器，而不产生实际代码。如果你想在 Linux2.0 上编译模块，这个宏应该在 *init_module* 中被使用。

EXPORT_SYMTAB；

如果你打算引出一些符号，那么模块必须在包含<linux/module.h>之前定义这个宏。

EXPORT_SYMBOL(name)；

这个宏表明你想引出这个符号名。它必须在任何函数之外使用。

EXPORT_SYMBOL_NOVERS(name)

使用这个宏而不是 EXPORT_SYMBOL()强制丢弃版本信息，即使是编译带有版本支持的代码。这对避免一些不必要的重编译很有用。例如，*memset* 函数将总以同样的方式工作；引出符号而不带版本信息允许开发者改变实现（甚至使用的数据类型）而不需 *insmod* 标出不兼容性。在模块化的代码中不大可能需要这个宏。

如果这些宏都没有在你的源码中使用，那么所有的非静态符号都被引出；这与在 2.0 中一样。如果这个模块是从几个源文件生成的，你可以从任何源文件引出符号，而且还可以在模块的范围中共享任何符号。

如你所看到的，引出符号表的新方法解决了一些问题，但这个创新也引入了一个重要的不兼容性：一个引出了一些符号的模块，如果想同时在 2.0 和 2.1 上编译运行，则必须用条件编译来包含两个实现。下面是 *export* 模块（*v2.1/misc-modules/export.c*）如何处理这个问题的：

（代码 384 #1）

上面的代码依赖于下面 *sysdep-2.1.h* 中的行：

（代码 384 #2）

当使用 2.1.18 或更新的核心时，REGISTER_SYMTAB 扩展为什么都不做，因为 *init_module* 中没有什么需要做的；在函数外使用 EXPORT_SYMBOL 是引出模块符号唯一需要做的。

声明参数

核心模块的新的实现利用了 ELF 二进制格式的特征以获得更好的灵活性。更特别地，当构造一个 ELF 目标文件时，你可以声明除“正文”、“数据”和“bss”之外的节。一个“节”是一个连续的数据区域，与“段”的概念类似。

对于 2.1，核心模块必须使用 ELF 二进制格式编译。事实上，2.1 核心利用了 ELF 的节（见“处理核心空间错误”），只能编译为 ELF。因此模块的限制并不是个真正限制。使用 ELF 允许信息域被存在目标文件中。好奇的读者可以使用 *objdump –section-headers* 来观察节头，用 *objdump –section=.modinfo –full-contents* 来查看模块特定的信息。实际上，.modinfo 一节是用来存储模块信息的节，包含被称做“参数”的值，可以在加载时修改。

当在 2.1 上编译时，一个参数可以用宏如下声明：

MODULE_PARM(variable, type-description)；

当你在源文件中使用这个宏时，编译器被告知在目标文件中插入一个描述串；这个描述表明 variable 是个参数，它的类型对应于 type-description。*insmod* 和 *modprobe* 查看目标文件，保证你被允许修改 variable，同时检查参数的实际类型。类型检查对防止不愉快的错误非常重要，例如用一个串覆盖了一个整数，或错把长整数当成了短整数。

按我的观点，讲述宏的最好办法时给出几行示例代码。下面的代码属于一个想象的网卡：

（代码 385）

type-description 串在头文件<linux/module.h>中被非常详细地介绍，并且为了你的方便，它可以在整个核心源码中找到。

值得给出的一个技巧是如何参数化一个数组的长度，象上面的 io。例如，设想网络驱动程序支持的外围板子的数目有宏 MAX_DEVICES 表示，而不是硬写入的数字 4。出于这个目的，头文件<linux/module.h>定义了一个宏（__MODULE_STRING），它用 C 预处理器将一个宏

“字符串化”。这个宏可以如下使用：

```
int io[MAX_DEVICES+1]={0,};
MODULE_PARM(io, “1-” __MODULE_STRING(MAX_DEVICES) “i”);
```

在前一行中，被“字符串化”的值与其他串接在一起构成目标文件中有意义的串。

scull 示例模块也用 MODULE_PARM 来声明它的参数（scull_major 和其他整数变量）。这在 Linux2.0 上编译时可能会出问题，那里这个宏未定义。我选择的简单的修正是在 *sysdep-2.1.h* 中定义 MODULE_PARM，这样在与 2.0 头文件编译时，它扩展为空语句。

其它有意义的值可以象 MODULE_AUTHOR()一样 存在模块的.*modinfo* 一节，但它们目前没有使用。请参考<linux/module.h>以获得更多的信息。

/proc/modules

/proc/modules 的格式在 2.1.18 中略有改变，而所有的模块化代码都被重写了。尽管这个改变并不影响源码，你可能对其细节不感兴趣，因为*/proc/modules* 在模块开发时经常被检查。

新格式和旧的一样是面向行的，每行包含下面的域：

模块名

这个域与 Linux2.0 相同。

模块大小

这是个十进制数，以字节为单位（而不是内存页）报告长度。

这个模块的使用计数

如果模块没有使用计数，这个计数报告-1。这是和新的模块化代码一道引入的新特征；你可以写一个模块，它的去除可以有一个函数控制而不是使用计数。这个函数判断模块是否能够被卸载。例如，*ipv6* 模块就使用这个特征。

可选标志

标志是文本串，每个都由括号包含，并由空格分隔。

参考本模块的模块列表

这个列表整体被包含在方括号内，表中的单个名字由空格隔开。

下面是*/proc/modules* 在 2.1.43 中的可能内容：

```
morgana% cat /proc/modules
ipv6                   57164   -1
netlink                 3180    0  [ipv6]
floppy                 45960    1    (autoclean)
monitor                  516    0    (unused)
```

在这个屏幕快照中，ipv6 没有使用计数，并依赖于 netlink；floppy 已经被 *kerneld* 加载，由“autoclean”标志给出，monitor 是我的一个小工具，控制一些状态灯，并在系统终止时关掉我的计算机。如你所看到的，它是“unused”，我并不关心它的使用计数。

文件操作

有几个文件操作在 2.1 里与 2.0 有不同的原型。这主要是出于处理大小不能放入 32 位的文件的需要。其不同由头文件 *sysdep-2.1.h* 处理，它根据使用的核心版本定义了几个伪类型。文件操作中引入的仅有的显著创新是 *poll* 方法，它用完全不同的实现代替了 *select* 方法。

原型的不同

四个文件操作表征一个新的原型；它们是：

```
long long (*llseek) (struct inode *, struct file *, long long, int);
long (*read) (struct inode *, struct fle *, char *, unsigned long);
long (*write) (struct inode *, struct file *, const char *, unsigned long);
int (*release) (struct inode *, struct file *);
```

它们在 2.0 中的对应者是：

```
int (*lseek) (struct inode *, struct file *, off_t, int);
int (*read) (struct inode *, struct file *, char *, int);
int (*write) (struct inode *, struct file *, const char *, int);
void (*release) (struct inode *, struct file *);
```

如你所见的，其不同在于它们的返回值（它允许了更大的范围），还有 count 和 offset 参数。头文件 *sysdep-2.1.h* 通过定义下面的宏处理这些不同：

read_write_t

这个宏扩展为参数 count 的类型以及 *read* 和 *write* 的返回值。

lssek_t

这个宏扩展为 *llseek* 的返回值类型。方法名字的改变（从 *lseek* 到 *llseek*）并不是个问题，因为你一般在 file_operations 中并不用名字对域赋值，而是声明一个静态结构。

lseek_off_t

lseek 的 offset 参数。

release_t

release 方法的返回值；或为 void 或为 int;

release_return(int return_value)

这个宏可以用来从 *release* 方法返回。它的参数用来返回一个错误代码：0 表示成功，负值表示失败。在比 2.1.31 老的核心中，这个宏扩展为 return，因为这个方法返回 void。

用前面的宏，一个可移植的驱动程序原型是：

```
lseek_t my_lseek(struct inode *, struct file *, lseek_off_t, int);
read_write_t my_read(struct inode *, struct file *, char *, count_t);
read_write_t my_write(struct inode *, struct file *, const char *, count_t);
release_t my_release(struct inode *, struct file *);
```

poll 方法

2.1.23 引入了 *poll* 系统调用，它是 system V 中 *select* 的对应者（由 BSD Unix 引入）。不幸的是不可能在 *select* 设备方法之上实现 *poll* 的功能，所以整个实现用不同的一个代替，它作为 *select* 和 *poll* 的后端。

在当前版本的核心中，file_operations 中的设备方法也叫 *poll*，与系统调用类似，因为其内部模仿这个系统调用。这个方法的原型是：

```
unsigned int (*poll) (struct file *, poll_table *);
```

驱动程序中设备特定的实现主要完成两个任务：

- 在一个可能在将来唤醒它的等待队列中将当前进程排队。通常，这意味着同时在输入和输出队列中对进程排队。函数 *poll_wait* 被用于这个目的，其工作方式与 *select_wait* 非常类似（细节请看第五章“增强的字符设备驱动程序操作”中“*select*”一节）。
- 构造一个位掩码描述设备的状态，并将其返回给调用者。这些位的值是平台特定的，在<linux/poll.h>中定义，它必须被包含在驱动程序中。

在讲述位掩码的每一位前，我想给出一个典型的实现。下面的函数是 *v2.1/scull/pipe.c* 的一部分，是*/dev/scullpipe* 的 *poll* 方法的实现。*scullpipe* 的内部在第五章介绍过。

（代码 389）

如你所看到的，这个代码相当简单。它比对应的 *select* 方法要容易。至于 *select*，状态位计为“ 可读 ”、“ 可写 ”，或“ 发生例外 ”（这是 *select* 的第三个条件）。

poll 各位的完全列表在下面给出。“ 输入 ” 位列在前面，然后是“ 输出 ”，一个“ 例外 ” 位列在最后。

POLLIN

如果设备可以被无阻塞地读，那么这个位必须被设置。

POLLRDNORM

如果“ 一般 ” 数据可以被读，这个位必须被设。一个可读设备返回（POLLIN | POLLRDNORM）。

POLLRDBAND

在目前的核心源码中这个位不被使用。Unix System V 使用这个位报告非 0 优先级的数据可读。数据优先级的概念与“ Streams ” 包相关。

POLLHUP

当一个读设备的进程看到文件结尾时，驱动程序必须设置 POLLHUP(挂起)。一个调用 *select* 的进程将被告知设备可读，这由 *select* 的功能说明。

POLLERR

设备上发生了一个错误条件。当 *poll* 被 *select* 系统调用调用时，设备被报告为既可读又可写，因为 *read* 或 *write* 将无阻塞地返回一个错误代码。

POLLOUT

如果设备可以被无阻塞地写，这个位在返回值中被设置。

POLLWRNORM

这个位与 POLLOUT 有相同的含义，有时甚至的确为同一个数。一个可写的设备返回（POLLOUT | POLLWRNORM）。

POLLWRBAND

与 POLLRDBAND 类似，这个位意味着非 0 优先级的数据可以被写到设备。只有 *poll* 的“ 数据报 ”实现用到着位，因为一个数据报可以传送“ 无团队数据（out-of-band data）”。*select* 报告设备是可写的。

POLLPRI

高优先级的数据（“ 无团队的 ”）可以被无阻塞地读取。这个位导致 *select* 报告文件上发生了一个例外条件，因为 *select* 将无团队包作为一个例外条件报告。

poll 的主要问题是它与 2.0 核心所使用的 *select* 方法没有任何关系。因此，处理这个不同的最好方法是使用条件编译来编译合适的函数，而且同时将它们都包含在源文件中。

如果当前版本支持 *select* 而不是 *poll*，那么头文件 *sysdep-2.1.h* 定义符号 __USE_OLD_SELECT__。这将你从在源码中必须引用 LINUX_VERSION_CODE 中解脱出来。v2.1 目录下的示例驱动程序使用了与下面类似的代码：

（代码 390）

（代码 391）

这两个函数用同样的名字调用，因为在结构 sample_fops 中 sample_poll 被引用，那里 *poll* 文件操作代替了 *select* 方法。

访问用户空间

核心的第一个 2.1 版引入了一种从核心代码访问用户空间的新（更好）方法。这个改变修正了一个长期存在的错误行为并增强了系统的性能。

当你位核心 2.1 编译代码，并需要访问用户空间时，你需要包含<asm/uaccess.h>，而不是<asm/segment.h>。你还必须使用一个与 2.0 不同的函数集。不用说，头文件 *sysdep-2.1.h* 尽可能地照顾了这些不同，允许你在 2.0 上编译时使用 2.1 的语义。

在用户访问中最令人注意的不同时 *verify_area* 没有了，因为多数验证都由 CPU 完成了。关于这个主题的细节见本章后面的“ 处理核心空间错误 ”。

可被用来访问用户空间的新的函数集是：

int access_ok(int type, unsigned long addr, unsigned long size);

如果当前进程被允许访问地址 addr 处的内存，函数返回真（1），否则为假（0）。这个函数取代 *verify_area*，尽管它进行较少的检查。和老的 *verify_area* 接收一样的参数，但是要快的多。在你复引用一个用户空间地址之前，这个函数应该被调用对之进行检查；如果你没有检查，用户有可能会访问和修改核心内存。本章后面的“ 虚拟内存 ”一节更细致地解释了这个问题。幸运的是，下面描述的大多数函数都替你进行了这个检查，因此你实际上并不需要调用 *access_ok*，除非你选择这样做。

int get_user(lvalue, address);

在 2.1 核心中使用的宏*get_user*与我们在 2.0 中使用的并不相同。其返回值在成功时为 0，否则为一个负的错误代码（总是-EFAULT）。这个函数的净效果是将从地址address取得的数据赋给*lvalue*。在通常的C语言含义中，这个宏的第一个参数必须是一个*lvalue*[*]。与 2.0 版中的这个函数类似，数据项的实际大小依赖于address参数类型。这个函数在内部调用*access_ok*。

int __get_user(lvalue, address);

这个函数完全类似 *get_user*，但它不内部调用 *access_ok*。当你访问一个已经从同一核心函数内部检查过的用户地址时，你应该调用*__get_user*。

get_user_ret(lvalue, address, retval);

这个宏是调用 *get_user* 的快捷方式，如果函数失败则返回 retval。

int put_user(expression, address);

int __put_user(expression, address);

put_user_ret(expression, address, retval);

这些函数与它们的 *get_*对应者非常类似，只是它们是向用户空间写，而不是读。成功时，值 expression 被写到地址 address。

unsigned long copy_from_user(unsigned long to, unsigned long from, unsigned long len);

这个函数从用户空间复制数据到核心空间。它代替旧的 *memcpy_tofs* 调用。这个函数内部调用 *access_ok*。返回值是未能传送的字节数。这样，如果发生错误，返回值必然大于 0；在那种情况下，驱动程序返回-EFAULT，因为错误是由错误的内存访问引起的。

unsigned long __copy_from_user(unsigned long to, unsigned long from, unsigned long len);

这个函数与 *copy_from_user* 一样，但它不内部调用 *access_ok*。

caopy_from_user_ret(to, from, len, retval);

这个宏是内部调用 *copy_from_user* 的快捷方式；如果失败，则从当前函数返回。

unsigned long copy_to_user(unsigned long to, unsigned long from, unsigned long len);

---

[*] 一个*lvalue*是一个可以作为赋值的左操作数的表达式。例如，count, v[34+check()], 和*（(prt+offset) ->field）是lvalue；i++,32, 和cli()都不是。

unsigned long __copy_to_user(unsigned long to, unsigned long from, unsigned long len);
copy_to_user(to, from, len, retval);

这些函数被用来将数据复制到用户空间，它们的行为非常类似于它们的 *copy_from* 的对应者。

2.1 版核心还定义了其它访问用户空间的函数：*clear_user*，*strncpy_from_user*，和 *strlen_user*。我不打算讨论它们了，因为 Linux2.0 中没有这些函数，并且驱动程序的代码也很少用到它们。有兴趣的读者可以看看<asm/access.h>。

使用新的接口

访问用户空间的新的函数集初看起来可能有点令人失望，但它们的确使程序员的日子好过的多了。在 Linux2.1 上，不再需要显式地检查用户空间；*access_ok* 一般不需要调用。使用新接口的代码可以直接进行数据传送。_ret 函数在实现系统调用时证明是相当有用的，因为一个用户空间的失败通常导致系统调用的一个返回-EFAULT 的失败。

因此，一个典型的 *read* 实现，看起来如下：

```
long new_read(struct inode *inode, struct file *filp, char *buf, unsigned long count);
{
/* identify your data (device-specific code) */
if (__copy_to_user(buf, new_data, count))
    return -EFAULT;
return count;
}
```

注意使用不进行检查的*__copy_to_user* 是因为调用者在把数据传输分派到文件操作之前已经检查了用户空间。这就象 2.0，*read* 和 *write* 不需要调用 *verify_area*。

类似地，典型的 *ioctl* 实现看起来如下：

```
int new_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
{
/* device-specific checks, if needed */
switch(cmd){
    case NEW_GETVALUE:
        put_user_ret(new_value, (int *)arg, -EFAULT);
        break;
    case NEW_SETVALUE:
        get_user_ret(new_value, (int *)arg, -EFAULT);
    default:
        return –EINVAL;
    }
return 0;
}
```

于版本 2.0 的对应者不同的是，这个函数在 switch 语句之前并不需要检查参数，因为每个 *get_user* 或 *put_user* 会进行检查。另一种实现方式如下：

（代码 394 #2）

另一方面，当你想写可以同时在 2.0 和 2.1 上编译的代码时，问题变得稍微复杂一些，因为在老的核心上，你不能用 C 预处理器伪装新的行为。你不能简单地#define 一个接收两个参数的 *get_user* 宏，因为实际的 *get_user* 实现在 2.0 中已经是个宏。

我在写既可移植有高效率的代码的选择是设置 *sysdep-2.1.h* 以提供具有下列函数的源码。下面只列出了读取数据的函数；写数据的函数行为完全一样。

int access_ok(type, address, size);

当在 2.0 上编译时，这个函数以 *verify_area* 的名义实现。

int verify_area_20(type, address, size);

通常，当为 Linux2.1 写代码时，你不需调用 *access_ok*。另一方面，当在 Linux2.0 上编译时，则必须调用 *verify_area*。这个函数就是要填平这个不同：当为 Linux2.1 编译时，它扩展为空；而为 2.0 编译时，它扩展为原来的 *verify_area*。这个函数不能被称做 *verify_area*，因为 2.1 已经有一个宏叫这个名字了。在 2.1 中定义的 *verify_area* 宏实现了 a*ccess_ok* 的老的语义，它的存在是为了简化源码从 2.0 到 2.1 的转换。（从理论上说，你可以在你的模块中留下 *verify_area*，只是将函数名改一下；这种简单移植技巧的缺点是新版本不能在 2.0 上编译。）

int GET_USER(var, add);

int __GET_USER(var, add);

GET_USER_RET(var, add, ret);

当在 2.1 上编译时，这些宏扩展为实际的 *get_user* 函数，即上面解释过的那些。当在 2.0 上编译时，*get_user* 的 2.0 实现被用来实现与 2.1 中同样的功能。

int copy_from_user(to, from, size);

int __copy_from_user(to, from, size);

copy_from_user_ret(to, from, size);

当在 2.0 上编译时，这些扩展为 *memcpy_fromfs*；而在 2.1 上，则使用本身的函数。*_ret* 一类在 2.0 上从不会返回，因为复制函数不会失败。

我个人比较喜欢这种实现兼容性的方法，但这并不是唯一的方法。在我的示例代码中，任何用户空间的访问（除了用来 *read* 或 *write* 的缓冲区，它们已经事先检查过了）之前，*verify_area_20* 必须被调用。另一种方法更加忠实于 2.1 的语义，即当用 2.0 时，在每个 *get_user* 和 *copy_from_user* 之前自动生成一个 *verify_area*。这个选择在源码级要更清晰一些，但在版本 2.0 上编译时效率相当低，包括代码大小和执行时间。

可以同时在 2.0 和 2.1 上编译的示例代码，如 *scull* 模块，可以在目录 *v2.1/scull* 中找到。我不觉得这个代码足够有趣，因此不在这里给出。

任务队列

从 2.1.30 开始的 Linux 版本不再定义函数 *queue_task_irq* 和 *queue_task_irq_off*，因为在 *queue_task* 上的实际加速不值得花精力维护两个独立的函数。当新机制被加到核心时，这就变得明显了。

在源码级，这是 2.0 和 2.1 之间唯一的区别；头文件定义了消失的函数简化了从 2.0 移植驱动程序。感兴趣的读者可以查看<asm/spinlock.h>以获得更多的细节。

中断管理

在 2.1 的开发中，有些 Linux 内部被修改了。新核心提供了对内部锁的很好的管理；通过使用几个细粒度的锁，而不是全局的锁，竞争条件被避免了，这样也获得了更好的性能----特别是 SMP 配置下。

更细的锁机制的一个结果是 intr_count 不再存在了。2.1.34 抛弃了这个全局变量，而布尔函数 *in_interrupt* 可以取而代之（这个函数从 2.1.30 开始存在）。目前，*in_interrupt* 是在头文件 <asm/hardirq.h>中声明的宏，这个头文件又包含在<linux/interrupt.h>中。头文件 *sysdep-2.1.h* 用 intr_count 的名义定义了 *in_interrupt* 以获得对 2.0 的向后兼容性。
注意虽然 *in_interrupt* 是个整数，intr_count 却是个 unsigned long，因此，如果你想打印这个值，并在 2.0 和 2.1 间可移植，你必须强制将这个值转换为一个显式的类型，并在调用 *printk* 时指定一个合适的格式。
在 2.1.37 中中断管理又引入了一个不同：快和慢中断处理程序不再存在了。SA_INTERRUPT 不被新版本的 *request_irq* 使用，但它在处理程序执行以前仍然控制着中断是否被打开。如果几个处理程序共享一个中断线，每个可以是个不同的“类型”。中断打开与否依赖于第一个被调用的处理程序。当中断处理程序存在时，下半部总是执行。

位操作

2.1.37 稍微改变了在<asm/bitops.h>中定义的位操作的作用。现在函数 *set_bit* 及其相关者返回 void，而新的类似 *test_and_set_bit* 的函数已被引入。新的函数集有如下原型：

```
void set_bit(int nr, volatile void * addr);
void clear_bit(int nr, volatile void * addr);
void change_bit(int nr, volatile void * addr);
int test_and_set_bit(int nr, volatile void * addr);
int test_and_clear_bit(int nr, volatile void * addr);
int test_and_change_bit(int nr, volatile void * addr);
int test_bit(nr, addr);
```

如果你想获得与 2.0 的后向兼容性，你可以在你的模块中包含 *sysdep-2.1.h*，并使用新的原型。

转换函数

版本 2.1.10 引入了一个新的转换函数，在<asm/byteorder.h>中声明。这些函数可以用来访问多字节值，只要这个值已知是小印地安字节序或大印地安字节序。因为这些函数为写驱动程序代码提供了很好的快捷方式，头文件 *sysdep-2.1.h* 在较早的版本就已经定义了它们。
2.1 核心源码提供的本身实现比 *sysdep-2.1.h* 提供的可移植的实现要快，因为它可以利用体系相关的功能。
新函数对应下面的原型，其中 le 表示小印地安字节序，be 表示大印地安字节序。注意编译器并不强制严格的数据类型化，因为大多数函数都是预处理宏；下面给出的类型仅供参考。

```
__u16 cpu_to_le16(__u16 cpu_val);
__u32 cpu_to_le32(__u32 cpu_val);
__u16 cpu_to_be16(__u16 cpu_val);
__u32 cpu_to_be32(__u32 cpu_val);
__u16 le16_to_cpu(__u16 le_val);
__u32 le32_to_cpu(__u32 le_val);
__u16 be16_to_cpu(__u16 be_val);
__u32 be32_to_cpu(__u32 be_val);
```

这些函数在处理二进制数据流时很有用（例如文件系统数据或存在接口板中的信息），版本 2.1.43 又增加了两个新的转换函数集。这些集允许你用指针获取一个值，或是对参数指定的

一个值进行就地转换。对应与 16 位小印地安字节序的函数又如下的原型；类似的函数对其它类型的整数也存在，导致一共 16 个函数。

```
__u16 cpu_to_le16p(__u16 *addr)
__u16 le16_to_cpup(__u16 *addr)
void cpu_to_le16s(__u16 *addr)
void le16_to_cpus(__u16 *addr)
```

“p”函数类似与指针的复引用，但在需要时转换这个值；“ s ”函数可以在原地转换一个值的印地安字节序（例如，cpu_to_le16s(addr) 和 addr=cpu_to_le16(*addr)完成的工作是一样的）。这些函数也在 *sysdep-2.1.h* 中定义了。为了避免双重解释的副作用，这个头文件用线入函数，而不是预处理宏。

vremap

在第七章“ 把握内存 ”中“ *vmalloc* 和朋友们 ”一节描述的 *vremap* 函数在版本 2.1 中得到一个新名字。新函数 *ioremap* 只是名字变了，它与旧的 *remap* 取同样的参数。响应的释放函数是 *iounmap*，它代替 *vfree* 来释放被重映射的地址。

这个改变是为了明确这个函数的实际作用：将 I/O 空间重映射到核心空间的一个虚地址。头文件 *sysdep-2.1.h* 强化了这种新规则，当在 2.0 版本编译时，它#define 了 *ioremap* 和 *iounmap* 到它们 2.0 的对应者。

虚拟内存

在核心的版本 2.1，Linux 的 Intel 移植对虚拟内存有了一个成熟的视图。早些的版本的内存管理一直使用“ 分段 ”的方法，这是从核心生命期的开始时期继承下来的。这个改变并不影响驱动程序代码，但不管怎样，还是值得一说的。

新的规则与 Linux 的其它移植匹配的起来。虚拟地址空间被构造成核心居于非常高的地址（从 3GB 往上），而用户地址空间在 0-3GB 范围。当一个进程运行在“ 管态 ”时，它可以访问两个空间。另一方面，当它运行在“ 用户态 ”时，它不能访问核心空间，因为属于核心的页被标记为“ 管理员 ”页，处理器阻止了对它们的访问。

这种内存布局有助于取消旧的 *memcpy_to_fs* 一类的函数，因为已经没有 FS 段了。核心空间和用户空间使用同一个“ 段 ”，其区别在于 CPU 所在的优先级。

处理核心空间错误

Linux 核心的 2.1 版本对从核心空间处理段错误的能力有一个极大的增强。本章里，我准备对其原则给一个快速的概述。新机制对源码的影响在“ 访问用户空间 ”中已经描述过。

如前面所提到过的，核心的最近版本充分利用了 ELF 二进制格式，特别是考虑到它的在编译的文件中定义用户定义的节的能力。编译器和链接器保证属于同一节的代码段在可执行文件中一定是连续的，因此当文件被装载时，在内存中也是连续的。

例外处理是通过在核心可执行映象(*vmlinux*)中定义两个新节实现的。每次当源码通过 *copy_to_user*, *put_user*, 或其读取的对应者访问用户空间时，一些代码被加到这两个节中。尽管这看起来是不可忽略的开销，这个新机制的一个结果是不再需要使用一个昂贵的 *verify_area*。而且，如果使用的用户地址是正确的，计算流将不会有一个跳转。

当被访问的用户地址是无效的时，硬件发出一个页面错。错误处理程序（在体系结构特定的

源码树中的 *do_page_fault*）确认这个错误是一个“ 不正确的地址 ”错（与“ 页不存在 ”相对），并使用下面的 ELF 节进行适当的动作：

__ex_table

这节是个指针对的表。每对的第一个指针指向一个可能因错误的用户空间地址而失败的指令，第二个值指向一个地址，处理器将在那里找到几条的指令来处理这个错误。

.fixup

这节包含指令，处理在__ex_table 中描述的所有可能的错误。这个表中每对的第二个指针指向居于.fixup 中的代码。

头文件<asm/uaccess.h>负责构造所需的 ELF 节。访问用户空间的每个函数（如 *put_user*）扩展为汇编指令，它将指针加到__ex_table 并处理.*fixup* 中的错。

当代码运行时，实际的执行路径有一下步骤组成：用于函数“ 返回值 ”的处理器寄存器被初始化为 0（也就是没有错误），数据被传送，返回知被传回调用者。一般的操作的确非常快。如果一个异常发生，*do_page_fault* 打印一条消息，查看__ex_table，跳转到.*fixup*，这里设置返回值为-EFAULT，然后跳转到访问用户空间的指令后位置。

新的行为可以用 *faulty*（在 *v2.1/misc-modules* 目录）模块来检验。*faulty* 在第四章“ 调试技巧 ”中“ 调试系统错误 ”一节描述。*faulty* 的设备结点通过读取一个短缓冲区界外来传送数据到用户空间，这样当读取一个在模块页以上的地址时，会导致一个页面错。有趣的是注意到这个错误依赖于使用核心空间中的一个不正确地址，而大多数情况下异常是有出错的用户空间地址造成的。

当在 PC 上用 *cat* 命令读 *faulty* 时，下面的消息被打印在控制台上：

```
read: inode c1188348, file c16decf0, buf 0804cbd0, count 4096
cat: Exception at [<c2807b7>](c2807115)
```

前一行是由 *faulty* 的 *read* 方法打印的，而后者是由错误处理程序打印的。第一个数字是错误指令的地址，而第二个是修正代码（在.*fixup* 节中）的地址。

其它改变

在 2.0 和 2.1.43 之间还有其它一些不同。以我的观点，它们不需要给予特别的关注，因此我将迅速地概述一下。

proc_register_dynamic 在 2.1.29 中消失了。最近的核心对每个*/proc* 文件使用 *proc_register* 接口 如果结构 proc_dir_entry 的 low_ino 域是 0，那么会被分配一个动态的 inode 号。当为 2.1.29 或更新的核心编译时，头文件 *sysdep-2.1.h* 象 *proc_register* 一样定义 *proc_register_dynamic*；这个在注册的 proc_dir_entry 结构以 0 为 inode 号时是可行的。

在网络接口驱动程序领域，*rebuild_header* 设备方法从 2.1.15 起有一个新的原型。如果你只开发以太网驱动程序，你不会关心这个不同，因为以太网驱动程序不实现它们自己的方法；它们依赖于通用的以太网实现。当旧的实现需要时，头文件 sysdep-2.1 定义了宏__USE_OLD_REBUILD_HEADER__。示例模块 *snull* 显示了如何使用这个宏，但每必要在这里给出。

网络代码的另一个改变影响了结构 enet_statistics，它从 2.1.25 起不再存在。代替它的是一个新结构 net_device_stats，它在<linux/netdevice.h>中定义，而不是<linux/if_ether.h>。新结构与旧结构类似，但是多了两个域存储字节计数器：unsigned long rx_bytes, tx_bytes；一个全特征的网络接口驱动程序应该与 rx_packets 和 tx_packets 一道增加这些计数器，尽管一个快速的计划可能要抛弃这些计数器。核心头文件将 enet_statistics（老结构的名字）定义为

net_device_stats（新结构的名字）以方便已有驱动程序的可移植性。
最后，我需要指出 current 不再是个全局变量 x86, Alpha,以及 Sparc 的核心移植使用了聪明的技巧将 current 存在处理器中。这样核心的开发者努力又挤出了几个 CPU 周期。这个技巧避免了大量的内存访问，有时还能释放一个通用目的寄存器；编译器经常分配处理器寄存器来高速缓存几个经常访问的内存位置，而 current 是经常访问的。在不同的移植中使用了不同的技巧以优化访问。Alpha 和 Sparc 版本使用一个处理器寄存器（编译器优化不使用的一个）来存储 current。而 Intel 处理器有有限数目的寄存器，编译器可以使用它们所有；在这种情况下技巧包括将结构 task_struct 和核心栈页存储在连续的虚存页内。这允许 current 指针被“编码”在栈指针中。对每个 Linux 支持的平台，头文件<asm/current.h>给出了实际选择的实现。
象所有重要的软件一样，Linux 一直在改变着。如果你想为这个最新的、最伟大的核心写驱动程序，你需要保持跟上核心的发展。尽管处理不兼容性看起来可能很困难，我们发现两点特性：首先，主要的程序设计技巧一直在那里，不太可能改变（至少不常）；第二，每次改变都变得更好了，经常使你在将来的开发中需要的工作越来越少。